UNIVERSITY
of
GLASGOW

Department of Computing Science

University of Glasgow

Lilybank Gardens

Glasgow, G12 8QQ

Species Trees and the Ultrametric Constraint

Neil Moore

CS4H

2006-07

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in electronic format.

Neil Moore .....

Contents

List of Figures

List of Code Listings

List of Tables

Abstract

"I start with a chicken. A good chicken. A cheap chicken wouldn't make a rich soup. And it has to have gray feathers"— **Elsie Zussman**

The supertree problem is of interest to biologists in their efforts to build a complete tree of life. It involves taking a set of input trees with (usually) overlapping leaf sets and building a supertree that contains all the input species and respects all the evolutionary relationships in the input trees. Although efficient imperative solutions to the problem exist, they are highly specialised to solving "pure" variations of the problem. For this reason, constraint programming solutions have been tried and these do indeed improve the flexibility of the solution as well as being more easily understandable. However this power comes at a cost in time and space efficiency. The space efficiency problems are most pressing, because modest input data from biological journals cannot be loaded into the memory of a powerful workstation.

The root of the space efficiency problem is that the *ultrametric* constraint is posted $C(n, 3)$ times in the species model, where n is the number of species being modelled. This cubic space complexity is compounded by the fact that naive implementations of the constraint have large constant factors. This project achieves this space improvement using a whole-matrix ultrametric constraint that reduces the overall space complexity by a factor of n to $O(n^2)$ and achieves 2 order of magnitude improvements in practice, even on quite small inputs.

Another outcome of the project was to undertake a detailed analysis of the ultrametric constraint, the end result of which is a constraint design maintaining generalised bounds arc consistency. A rigorous proof of the correctness of this constraint was completed. Other designs and implementations of the same constraint were produced. The set of constraints were compared in an empirical study.

Preliminary results indicate that this level of consistency is sufficient to avoid search altogether, and this leads to a worst case time bound of $O(n^4)$ to solve the supertree problem, compared to $O(n^{n^2})$ for previous designs.

Next, a generalised ultrametric constraint was designed to provide the same time and space benefits to an alternative supertree model incorporating ancestral divergence dates. This constraint was also proved correct rigorously but more informally.

The report starts with reviews of the fields of constraint programming and species trees. No knowledge of either subject is assumed, although some programming knowledge and mathematical maturity are necessary.

Chapter 1

Background in constraint programming

"Consistency is the last refuge of the unimaginative."— **Oscar Wilde**

1.1 The Constraint Satisfaction Problem

The presentation in this section is based on [Smi01] but similar material can be found in the first section of almost any publication about constraint programming.

A *finite domain* CSP (henceforth, CSP) consists of a set of *variables* $\{v_1, \ldots, v_n\}$, each with a *domain* d_i that comprises a (finite) set of possible values, plus a set C of *constraints* that restrict the values that variables can simultaneously take. The motivation for modelling a problem as an instance of the CSP is to find zero or more *solutions* which are simultaneous assignments of domain values to all variables in such a way that all constraints are satisfied.

Domains commonly contain sequences of consecutive integers, though they can also be arbitrary sets of integers and mathematical entities like sets and graphs[DDD05].

Constraints may be *unary*, meaning that they constrain one variable; *binary*, meaning that they constrain two; *ternary*, three; and more generally *n-ary*. The formal definition of a constraint is that it is a subset of the cartesian product of the domains of the variables that it is over, this subset specifies the values that are allowed together (although the alternative of specifying the values that are not allowed is equally descriptive). In practice, a constraint may be specified *extensionally*, where the acceptable subset of the cartesian product is listed exhaustively, or *intensionally*, where the subset is described by some well-defined relation between the values.

An example of a constraint that would be specified extensionally is one that constrains variables representing consecutive seats around a dinner table to contain values corresponding to people from different couples. The definition is so nebulous that we could not expect to characterise the inconsistent values any more concisely than by a list, e.g.

$$\begin{array}{lll} \{ & \text{(Liz,} & \text{Philip),} \\ & \text{(David,} & \text{Victoria),} \\ & \text{(Elton,} & \text{David)} \quad \} \end{array}$$

In constrast, the $\leq$ constraint's extension would occupy $O(1 + 2 + \ldots + n) = O(n^2)$ space for two equal variables but it is very easy to verify if a pair of values conforms to it—certainly a lot easier than searching a data structure for the pair.

Since all constraints presented in this report have $\Omega(n^3)$[1] extensions and satisfactory intensional solutions, we will consider extensional constraints no further.

Now that constraints are defined we can be a little more rigorous in defining a solution to a CSP. A solution to a CSP with variables $x_1, \ldots, x_n$ is a tuple $(a_1, \ldots, a_n)$ that *satisfies* all the constraints C. A solution satisfies a k-ary constraint $c \in C$ over variables $x_{s_1}, \ldots, x_{s_k}$ whenever $(a_{s_1}, \ldots, a_{s_k}) \in c$.

A useful way to think about a CSP is as a graph whose vertices are variables and whose edges are constraints between variables[2].

1.2 The CSP in practice

Many problems of practical interest can be characterised as CSPs. To solve such problems by computer a CSP *solver* program is told about all the variables, domains and constraints that characterise the problem (the *model*). The solver will then provide solutions, if any exist.

One particular way of doing this is by using the JCHOCO package for JAVA. An outline procedure for creating and solving a problem in JCHOCO is as follows:

1. Create a new `Problem` object to represent the model. It provides methods to create variables; create and post constraints; instigate and guide the search process; and access solutions.
2. Create the variables in the problem, and simultaneously specify their domains (e.g. `prob.makeBoundIntVar("x",0,10)`).
3. Post constraints over the variables (e.g. `prob.post(prob.equals(x,y))`).
4. Tell the solver to find one or all solutions, and output them (e.g. `prob.solve(true)`).

The beauty of the JCHOCO system is that the ubiquitous OO paradign and the declarative paradigm co-exist, enhancing the power and readability of OO code and adding flexibility to constraint modelling by the addition of control structures and full IO facilities to generate models programmatically (see Section 1.2.2 for an example of these techniques). Neither of these features are essential in a CSP solver, for example the MINION[GJM06] solver takes as input a text file specifying the variables and constraints; any variety must be generated externally.

We will now try basic modelling in JCHOCO by an easy example and then a more difficult one.

1.2.1 A PIN number puzzle

Problem: To find the unique 4 digit bankcard PIN *abcd* such that each digit is different, the two digit number *ab* is 3 times the number *cd* and *bc* is 2 times *da*.
Model: 4 variables a, b, c, d each with domain $0, \ldots, 9$ to represent each of the 4 digits in the result. Constraints $\{a \neq b, a \neq c, a \neq d, b \neq c, b \neq d, c \neq d\}$ to make the digits different and constraints $\{10a + b = 3(10c + d), 10b + c = 10d + a\}$ to enforce the digit relationships.
Solution: See Listing 1.1.
Commentary:

[1] The big omega Ω notation means the opposite of big oh O. That is that a function is $\Omega(f(n))$ if it grows asymptotically *at least* as fast as $f(n)$.

[2] Assuming binary constraints throughout, for n-ary constraints the graph is a hypergraph (see [Ros98])

Line 3 Create `Problem` to encapsulate model and to direct solver.

Line 6 Create variable `a` whose domain is $0, \ldots, 9$.

Line 12 Create and post a constraint that $a \neq b$.

Lines 20-21 Create and post a constraint that $10c + 1d = 30a + 3b$.

Line 26 Tell the solver to search for all solutions (`false` parameter means to stop after finding the first).

Lines 27-29 Print a "pretty" representation of each solution.

Line 30 Output the number of solutions.

Output: See Listing 1.2. This lists the values assigned to variables in the solution found ($a = 2$, $b = 1$, ...) as well as the fact that only one solution was found.

1.2.2 Crystal maze puzzle

The PIN puzzle was a single problem, whereas the crystal maze puzzle[3] is a template for an infinite number of problems. The specification for a problem is provided in a text file. This means that Java's IO facilities and programming structures must be used to read it in and post it in a different way each time.

Problem: Given graph $G = (V, E)$ and $|V| = n$, find a unique labelling l for the vertices such that $(u, v) \in E \Rightarrow |l(u) - l(v)| \neq 1$, i.e. neighbouring vertices labelled non-consecutively.

Model: For each vertex v_i create a variable x_i whose domain is $1, \ldots, n$. Constrain variables to be pairwise different and $|x_i - x_j| \neq 1$ for vertices v_i and v_j joined by an edge.

Instance: An instance with 3 vertices and 1 edge (v_1, v_2) is specified by a file containing:

```
3 1
1 2
```

Solution: See Listing 1.3.

Commentary:

Lines 14-15 Create an array of `numNodes` variables each with domain $1, \ldots,$ `numNodes`.

Lines 17-25 There's nothing new in the individual lines of code here, but the way that the constraints are posted differently depending on the data in the file is new. This sort of technique is how a single constraint program can be made to solve different problems.

Line 28 Post the constraint that all the variables must take different values. This has the possibility of doing more propagation than a clique of $\neq$ constraints[Rég94].

Output: See Listing 1.4, where two distinct solutions were found and both have been printed out.

[3]Despite its name, this puzzle has nothing to do with crystal mazes—it was a challenge on the TV gameshow of the same name and has come to be known by this moniker.

```
1  public class CSolve {
2      public static void main(String[] args) throws ContradictionException {
3          Problem pin = new Problem();
4
5          //variables to represent each individual digit
6          IntVar a = pin.makeEnumIntVar("a", 0, 9);
7          IntVar b = pin.makeEnumIntVar("b", 0, 9);
8          IntVar c = pin.makeEnumIntVar("c", 0, 9);
9          IntVar d = pin.makeEnumIntVar("d", 0, 9);
10
11         //all different constraint
12         pin.post(pin.neq(a,b));
13         pin.post(pin.neq(a,c));
14         pin.post(pin.neq(a,d));
15         pin.post(pin.neq(b,c));
16         pin.post(pin.neq(b,d));
17         pin.post(pin.neq(c,d));
18
19         //relationship between digits, using scalar product expressions
20         pin.post(pin.eq(pin.scalar(new int[] {10, 1}, new IntVar[] {c, d}),
21                         pin.scalar(new int[] {30, 3}, new IntVar[] {a, b})));
22         pin.post(pin.eq(pin.scalar(new int[] {10, 1}, new IntVar[] {d, a}),
23                         pin.scalar(new int[] {20, 2}, new IntVar[] {b, c})));
24
25         //find and output all solutions
26         pin.solve(true);
27         do {
28             System.out.println(pin.pretty());
29         } while(pin.nextSolution().booleanValue());
30         System.out.println(pin.getSolver().getNbSolutions() + " solutions");
31     }
32 }
```

Listing 1.1: Code to solve PIN problem

```
1  $$ java -cp ~/project:. CSolve
2  Pb[4 vars, 8 cons]
3  Pb[4 vars, 8 cons]
4  ==== VARIABLES ====
5  a:2{2}
6  b:1{1}
7  c:6{6}
8  d:3{3}
9  Pb[4 vars, 8 cons]
10 ==== CONSTRAINTS ====
11 a:2 != b:1
12 a:2 != c:6
13 a:2 != d:3
14 b:1 != c:6
15 b:1 != d:3
16 c:6 != d:3
17 10*c:6 + 1*d:3 + -30*a:2 + -3*b:1 = 0
18 10*d:3 + 1*a:2 + -20*b:1 + -2*c:6 = 0
19
20 1 solutions
```

Listing 1.2: Result for PIN problem

```
1  public class Crystal {
2      public static void main(String[] args) throws IOException,
3                                                     ContradictionException {
4          StreamTokenizer st =
5              new StreamTokenizer(new InputStreamReader(System.in));
6          //get number of nodes
7          st.nextToken(); int numNodes = (int)st.nval;
8
9          //get number of edges
10         st.nextToken(); int numEdges = (int)st.nval;
11
12         //put together variables so we can create constraints as we read edges in
13         Problem pb = new Problem();
14         IntDomainVar[] nodes =
15             pb.makeBoundIntVarArray("nodes", numNodes, 1, numNodes);
16
17         //read in constraint and post non-consecutive constraint on the fly
18         for(int i = 0; i < numEdges; i++) {
19             st.nextToken(); int n1 = (int)st.nval - 1;
20             st.nextToken(); int n2 = (int)st.nval - 1;
21             IntExp sub = pb.minus(nodes[n1], nodes[n2]);
22             //post non-consecutive constraint
23             pb.post(pb.neq(sub, 1));
24             pb.post(pb.neq(sub, -1));
25         }
26
27         //ensure that all numbers are accounted for
28         pb.post(pb.allDifferent(nodes));
29
30         //now solve and output
31         pb.solve(true);
32         do {
33             System.out.println(pb.pretty());
34         } while(pb.nextSolution().booleanValue());
35         System.out.println(pb.getSolver().getNbSolutions() + "␣solutions");
36     }
37 }
```

Listing 1.3: Code to solve Crystal Maze puzzle

```
1  $$ java -cp ~/choco:. Crystal < trivinst
2  Pb[3 vars, 3 cons]
3  Pb[3 vars, 3 cons]
4  ==== VARIABLES ====
5  nodes[0]:1[1, 1]
6  nodes[1]:3[3, 3]
7  nodes[2]:2[2, 2]
8  Pb[3 vars, 3 cons]
9  ==== CONSTRAINTS ====
10 nodes[0]:1 != nodes[1]:3 + 1
11 nodes[0]:1 != nodes[1]:3 - 1
12 choco.global.BoundAllDiff@18558d2
13
14 Pb[3 vars, 3 cons]
15 Pb[3 vars, 3 cons]
16 ==== VARIABLES ====
17 nodes[0]:3[3, 3]
18 nodes[1]:1[1, 1]
19 nodes[2]:2[2, 2]
20 Pb[3 vars, 3 cons]
21 ==== CONSTRAINTS ====
22 nodes[0]:3 != nodes[1]:1 + 1
23 nodes[0]:3 != nodes[1]:1 - 1
24 choco.global.BoundAllDiff@18558d2
25
26 2 solutions
```

Listing 1.4: Result for Crystal Maze puzzle

1.3 Implementation of constraint solvers

The implementation of solvers for the CSP is an interesting and varied subject which we will touch on only briefly and only then when it is relevant to the aims of the project.

1.3.1 Search techniques: Generate and test, forward checking and maintaining arc consistency

JChoco uses one out of several available search techniques to find solutions. It has the key properties of *completeness* and *soundness*, meaning that all solutions are found (completeness) and that all found solutions are correct (soundness).

We will begin by looking at some legacy search techniques (all of which are both sound and complete) before moving on to the modern technique that JChoco employs.

The *generate and test* algorithm works by repeatedly instantiating every variable with a value from its domain and checking if all constraints are satisfied. If so, the instantiation is a solution; if not, it is discarded. Once every instantiation has been tried the search process is finished. This procedure has $O(\underbrace{m.m\ldots m}_{n \text{ times}}) = O(m^n)$ time complexity for n variables each with m values[Smi01]. Intuitively, the reason that this algorithm has been superseded is that it blindly continues instantiating even when the partial solution is incorrect. For example, with variables x, y and z and constraint $x \neq y$ if $x \leftarrow 1$ and $y \leftarrow 1$ the algorithm will proceed to try instantiating z although it can never succeed in producing a solution. With more variables or larger domains this could result in a great deal of unnecessary work and is known as *thrashing*[4].

Backtracking algorithms like backmarking, backjumping and conflict-directed backjumping (see [Pro93]) exploit the above observation by not continuing with instantiation once a partial assignment violates any constraint.

A further refinement is to infer from currently instantiated variables what future variables cannot be instantiated to, thereby pruning subtrees from the search tree ahead of time. Algorithms that do this are categorised as *forward checking*[HE80]. A simple method for forward checking (usually known simply as forward checking or FC) is to remove from future domains values that are directly incompatible with each new instantiation. However, the *maintaining arc consistency* (MAC)[SF94a] algorithm is a smarter method that not only does this but also cascades removals so that all remaining values in all domains are pairwise compatible, rather than only past and future variables being compatible. To see the difference between these consider variables x, y and z each with domain $\{1, 2, 3\}$, and constraints $x \neq y$ and $y = z$. If the search process has $x \leftarrow 1$ then with plain FC the remaining domains are pruned to $\{2, 3\}$ for y and $\{1, 2, 3\}$ for z, that is, z's domain is unchanged. Contrast this with MAC where as before y's domain is $\{2, 3\}$ but this time z's domain is trimmed down to $\{2, 3\}$. This extra trimming is due to the effect of cascading the loss of $1 \in y$.

The precise sense in which variables are compatible with one another is defined by *arc consistency* (AC) and in maintaining arc consistency every pair of variables is kept AC throughout the search process[5]. Two variables v_i and v_j with domains d_i and d_j are AC w.r.t. a constraint C if and only if $\forall x \in d_x$, $\exists y \in d_y$ s.t. $(x, y) \in C$ and $\forall y \in d_y$, $\exists x \in d_x$ s.t. $(x, y) \in C$. This is not the only only possible level of consistency, more will be described in Section 1.3.3.

[4] C.f. The idea of thrashing in virtual memory where virtual memory pages are loaded and unloaded intensively.

[5] Here the AC acronym is being overloaded to mean arc consistent rather than arc consistency. It should be clear by the context what the intended meaning is.

Just because a problem is AC doesn't mean it has a solution. Higher levels of consistency like SAC[PSW00] can rule out more non-solutions ahead of time but potentially with a time penalty. Experimental evidence suggests[SF94b] that AC is a good trade-off and so we will not consider higher consistency levels any further.

Arc consistency can be maintained in many different ways. In JChoco a particular specialisation of the AC-5 algorithm[HDT92] is used, but it will also be necessary to describe another specialisation called AC-3[6]. These AC algorithms are the subject of the next Section 1.3.2.

1.3.2 Arc consistency algorithms

The definition of AC says that all values should be *supported* by all constraints. That is, for any x and constraint c, there is a y that partners x to satisfy c. In practice this means that AC algorithms must remove unsupported values from the domains of variables where necessary. In the MAC algorithm this is done before search as well as during search whenever a domain is reduced. In this section we will see AC-3 and AC-5 which are both used to implement MAC, so that they will somehow keep a problem AC as search proceeds.

Notice that when a variable is reduced only other variables connected to it through a chain of constraints can possibly be reduced as a result of enforcing AC (i.e. AC is a *local property*). A corollary of this local property is that when a variable's domain changes during the search process we need only check variables that share a constraint with it to ensure all their values are still supported rather than checking everything. This observation will be important in understanding both AC-3 and AC-5.

1.3.2.1 AC-3

The AC-3 algorithm[Mac75] uses a set S of directed edges to keep track of variables that may not be AC with a neighbour. For example if the set contains (v_i, v_j) this means that v_i may need to be revised due the loss of a value from v_j.

Let C_{ij} be the constraint between v_i and v_j, if it exists[7]. The AC-3 algorithm is to repeatedly remove a pair (v_i, v_j) from S until none remain. Each time set $D = \{x : x \in v_i \wedge \neg\exists y \in v_j \text{ s.t. } (x, y) \in C_{ij}\}$ is found, i.e. the set of all values in v_i that are unsupported by v_j. These values D are then removed from the domain of v_i to make it AC w.r.t. v_j. Next, $\forall C_{ki} \in C$, C_{ki} is added to S, so that the effects on all connected variables of the removals in v_i will be investigated. Once S becomes empty AC has been established amongst all variables.

When establishing AC for the first time before search starts, the set will contain every edge in both directions wherever a constraint exists. When a variable is instantiated during search and AC is to be re-established, only edges ending at the instantiated variables are added to the set before running the algorithm.

Let m be the size of the largest domain and let c be the number of constraints. The time complexity of discovering the set D is $O(m^2)$. To make the whole problem AC S starts with $2c$ pairs in it and, since in the worst case each set D could have size 1, up to $2cm$ further pairs may have to be added. Hence the algorithm has overall $O((2c + 2cm)m^2) = O(cm^3)$ worst case time complexity[YY01].

[6] AC-3 was not conceived as a specialisation of AC-5at first[Mac75], but was retrofitted to be such[HDT92]

[7] If more than one such constraint exists then let it be the union of the two constraints. Furthermore, C_{ji} is the same as C_{ij} but with the tuples in the constraint reversed.

1.3.2.2 AC-5

The AC-5 algorithm[HDT92] makes the observation that when particular values are lost from a variable, consistency algorithms may be able to take advantage of the specific value that has been lost to speed up pruning. For example if $x = y$ and x loses the value 1 then we know for a fact that 1 is the only value ruled out for y. In the general case the complexity of AC-5 is $O(cm^3)$ which is no better than AC-3. However, in special cases the worst case for enforcing AC can be reduced to O(cm) (for justification of this see [HDT92]).

In AC-5, instead of a set of directed edges to process, we have a set S of triples (v_i, v_j, y) each of which means that "as a result of the loss of y from v_j, v_i must be tested for AC". Once all such triples have been processed the problem is AC.

To establish AC for the first time before search, the algorithm first checks each variable v_i in turn for consistency with its neighbours and obtains a set Δ_1 which is all the values that should be removed from v_i. Now $\{(v_k, v_i, d) : C_{ik} \in C \wedge d \in \Delta_1\}$ is added to S, so that the effect on other variables of these removals from v_i is considered. The next stage is to repeatedly remove one triple (v_i, v_j, y) from S, and to carry out any removals Δ_2 from v_i needed as a result. Now extra triples $\{(v_k, v_i, d) : C_{ik} \in C \wedge d \in \Delta_2\}$ are added to S to propagate the effects of these removals to neighbouring variables. Once the queue is empty the changes have settled down and the problem is AC.

During search triples $\{(v_i, v_j, y) : C_{ij} \in C\}$ are added to S when variable v_j loses value y, so that only the minimum work is done to re-establish AC.

For a more detailed discussion of the operation of AC-5 and a proof of correctness see [HDT92].

1.3.2.3 General discussion

For the purposes of this project it is only necessary to know that AC-5 gives constraint designers the chance to make inferences based on the particular value removed from a domain, rather than being limited to just knowing that *some* value has been lost, as is the case for AC-3. Also, the fact that AC-5 is complete and sound[HDT92] means that the contract on a custom constraint (such as those presented in this report) is that when it is notified of the loss of a value, everything that it removes is genuinely unsupported. Without this guarantee search could be incomplete. Furthermore if a constraint allowed incorrect pairs to be instantiated, search could be unsound.

1.3.3 Other levels of consistency

So far we have assumed that variables can have arbitrary domains. For a domain with size d implementations of variables will require $\Omega(d)$ space. However by restricting attention to domains containing continuous sequences of values, the space complexity can be reduced to O(1). This is done by storing a current lower bound and current upper bound. Clearly this representation cannot have arbitrary non-bound values removed and hence it may be impossible to maintain full AC.

For such occasions there exists an analog of AC for bound variables called *bounds AC*, this means that rather than having to support every value in a domain, it is sufficient that the upper and lower bounds are supported. In JChoco it is possible to mix AC and bounds AC constraints and as a result of this the whole problem may not be kept fully AC by the MAC algorithm. Nevertheless, final solutions will be correct because once a variable has been instantiated its lower and upper bounds are equal and the definition of bounds AC becomes equivalent to the definition of AC.

As an example of the difference suppose that a bounds AC = (equals) constraint is used. The CSP $x = \{1, 3\}$, $y = \{1, 2, 3\}$ and $x = y$ is bounds AC but not AC.

The choice of whether to use bounds AC or AC is closely related to the choice between bound variables (with sequential domains) and enumerated variables (with arbitrary domains). Choice of bound variables may be justified due to space concerns. Also, as above, full AC may not be possible with bounds variables, and the use of a specialised bounds AC constraint on such variables could reduce running time. Another justification for using bounds AC is possible reduction in running time if lazyness at nodes dominates an increase in search tree size.

The species model that this project is based around uses bounds variables and so many of the custom constraints in this project enforce bounds AC. It would be interesting future work to investigate how using enumerated variables and full AC affects results.

1.3.4 Augmenting the search process

A common modelling technique is to introduce auxiliary variables whose values are irrelevant but constrained in such a way that an inconsistency in a solution manifests itself as a domain wipeout. For this reason and others a constraint modeller may wish to limit the variables that are instantiated to only those that are part of a solution and to leave out the auxiliary variables. In a constraint model, the values that are actually being searched are called the *search variables*. In JCHOCO, by default, all variables are search variables but this can be changed.

Until now we have tacitly assumed that variables are instantiated in an arbitrary fixed order, but this does not need to be the case. Some constraint toolkits, and JCHOCO in particular, offer *dynamic variable ordering heuristics* that are intended to reduce search effort by making an intelligent choice of the order in which to instantiate variables. As an example of a situation where this could be beneficial, consider variables x, y and z with domains $\{1, \ldots, 1000\}$, $\{1, \ldots, 1000\}$ and $\{1000\}$ respectively, with constraints $x = z$ and $y = z$. Using MAC and a static variable order of x, y, z, 1000 instantiations $x \leftarrow 1$, $x \leftarrow 2$, $\ldots$, $x \leftarrow 1000$ will have to be attempted before a value for x is found that is consistent with the domain of z. We then have the same problem for variable y. A dynamic solution that would have worked well in this instance would have been to choose to next instantiate the variable whose domain is smallest, because in this case the domains of x and y would immediately have been reduced to $\{1\}$. This is known as the *min-domain* heuristic and it is the default in JCHOCO. Intuitively, it works well because if a solution is impossible it will be discovered sooner by failing on the smallest domain. This is an example of the fail first heuristic[HE80].

1.4 Implementation of custom constraints in JCHOCO

In JCHOCO, constraints are represented by subclasses of `AbstractIntConstraint`. This class has a large number of methods, many of which can be overridden if desired, but to implement a custom constraint only a small subset of these have to be implemented[8]:

[8]It is not clear from JCHOCO documentation which methods need to be overridden but these ones seem to work fine and Patrick Prosser has established that they are sufficient by personal correspondence with the authors of JCHOCO. Strictly speaking it should be sufficient to implement `awakeOnRem()` and `awake()`, because the former is analogous to the operation of removing a triple from the set S in AC-5 and the latter to checking every value for support at the outset. It is outwith the scope of this project to investigate the issue and it makes little difference to the correctness of the constraint's designs, even if the uncertainty reduces confidence in implementations a little bit.

`awake()` Invoked by the MAC algorithm before search to make the constraint AC.

`awakeOnRem(idx, val)` Invoked during search to re-establish consistency after variable index `idx` has lost value `val`.

`awakeOnRemovals(idx, valIter)` Invoked during search to re-establish consistency after variable index `idx` has lost all the values in `valIter`.

`awakeOnInst(idx)` Invoked during search to re-establish consistency after variable index `idx` has been instantiated.

`awakeOnInf(idx)` Invoked during search to re-establish consistency after variable index `idx` has lost one or more values causing a change in its lower bound.

`awakeOnSup(idx)` Analogous to `awakeOnInf(idx)` for upper bound change.

Generally speaking, the bodies of these functions will contain code that looks at the current domains of variables that the constraint acts over and then removes some unsupported values from variables in the constraint. Whenever a constraint empties a variables domain, it will (directly or indirectly) throw a `ContradictionException` that results in backtracking.

The generic AC-5 algorithm only notifies constraints when individual values are lost. In JChoco the above special cases are identified to make the job of writing constraints easier. We will see an example in the next section of a constraint that doesn't need to do any propagation when lower bounds are changed and in this case `awakeOnInf()` would be a "do nothing" method to save itself unnecessary work. Furthermore if a lower bound changes multiple times before the change is propagated, JChoco may coalesce these notifications into one function call to `awakeOnInf()`. Cases for other methods are similar.

When a constraint makes changes to a variable's domain, the underlying JChoco library adds triple to the AC-5 queue to cascade the change, by notifying all the other constraints on that variable. Hence each constraint does its own job in isolation to the others; they need not notify each other of domain event directly.

1.4.1 An example constraint in JCHOCO

An implementation of the $<$ (strictly less than) constraint is shown in Listing 1.5. It illustrates most of the basic techniques of programming custom constraints in JChoco:

Line 3 Notify the superclass (and indirectly the AC-5 algorithm) of the variables involved in the constraint.

Line 4 Create convenient aliases for the variables involved.

Lines 6-9 An increase in the lower bound of `v1` cannot result in loss of support for any value. This is because any value v in `v0` that was supported by the lower bound must still be supported by a larger value ($v <$ v1.inf $<$ anything else in `v1`). When `v0`'s lower bound is changed only values in `v1` that are greater than the new lower bound are still supported, this change in the `v1`'s domain is achieved by changing its lower bound (line 8).

Lines 10-13 Symmetric with `awakeOnInf()`.

Lines 14-19 The reasoning here is a special case of `awakeOnInf()` and `awakeOnSup()`. When `v0` is instantiated everything in `v1` must now be bigger than it. When `v1` is instantiated everything in `v0` must be smaller than it.

```
1  class NLT extends AbstractBinIntConstraint {
2      public NLT(IntDomainVar v0, IntDomainVar v1) {
3          super(v0, v1);
4          this.v0 = v0; this.v1 = v1;
5      }
6      public void awakeOnInf(int idx) throws ContradictionException {
7          if(idx == 0 && v1.getInf() < v0.getInf() + 1)
8              v1.setInf(v0.getInf() + 1);
9      }
10     public void awakeOnSup(int idx) throws ContradictionException {
11         if(idx == 1 && v0.getSup() > v1.getSup() - 1)
12             v0.setSup(v1.getSup() - 1);
13     }
14     public void awakeOnInst(int idx) throws ContradictionException {
15         if (idx == 0)
16             v1.setInf(v0.getVal() + 1);
17         else
18             v0.setSup(v1.getVal() - 1);
19     }
20     public void awakeOnRem(int idx, int a) throws ContradictionException {
21         if(idx == 0 && a < v0.getInf())
22             awakeOnInf(idx);
23         else if(idx == 1 && v1.getSup() < a)
24             awakeOnSup(idx);
25     }
26     public void awakeOnRemovals(int idx, IntIterator deltaDomain)
27         throws ContradictionException {
28         while(deltaDomain.hasNext()) {
29             awakeOnRem(idx, deltaDomain.next();
30     }
31     public void awake() throws ContradictionException {
32         awakeOnInf(0); awakeOnSup(1);
33     }
34 }
```

Listing 1.5: A JChoco constraint for $<$

Lines 20-25 When an individual value `a` is lost from `v0` then it can only cause the loss of support for a value in `v1` when it was a lower bound. In this case the code for `awakeOnInf(0)` does the correct propagation. The reasoning for a loss from `v1` is symmetric.

Lines 26-30 No special reasoning for removal of multiple values, just repeatedly apply procedure for removing one value.

Lines 31-33 Only insufficiently low values in `v0` and insufficiently high values in `v1` may be unsupported. These will be clipped by the procedures in `awakeOnInf()` and `awakeOnSup()`.

Chapter 2

Background in species trees

"A tree's a tree. How many more do you need to look at?"— **Ronald Reagan**

"If a frog turned into a monkey, shouldn't you have lots of fronkies?"— **Bishop Wayne Malcolm**

2.1 Trees

A *tree* is a connected, acyclic, undirected graph. A *rooted tree* is a tree in which there is a distinguished node called the *root*. Let x and y be nodes in a rooted tree T with root r. A node is an *ancestor* of x if and only if it lies on the unique path from r to x. Conversely y is a *descendent* of x if and only if x is an ancestor of y. If y follows x on some path from the root then y is x's *child* and x is y's *parent*. Hence only the root node has no parent. A node with no children is called a *leaf*, anything else is an *internal node*. The *degree* of a node is the number of children it has. The *height* of a tree T is the length of the longest path in the tree. The *depth* of x is the length of the path to it from the root. The *most recent common ancestor* (m.r.c.a.) of x and y is the node with greatest depth that is an ancestor of both x and y. In this definition of trees the children are implicitly *unordered*. For an illustration of all the above concepts see Figure 2.1.

A bifurcating tree is a tree in which every internal node has degree 2, such a tree is shown in Figure 2.2.

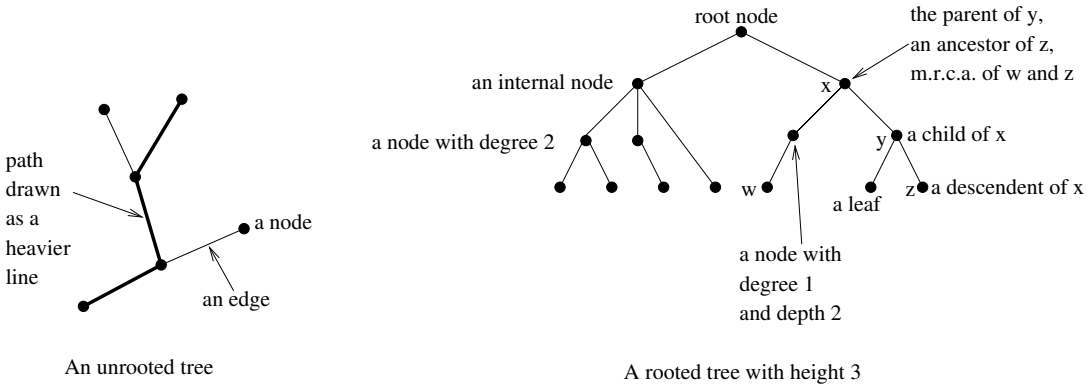


Figure 2.1: Illustrations of trees and related definitions

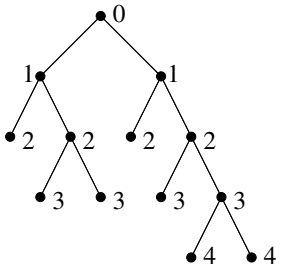


Figure 2.2: A bifurcating tree labelled by node depth.

2.2 Ultrametrics

A *metric*[Wei07a] is a nonnegative function $g(x, y)$ that provides a sort of "distance" between two values x and y in a set S. It must

- satisfy the triangle inequality that $\forall x_1, x_2, x_3 \in S, \exists$ a permutation $x_{p_1}, x_{p_2}, x_{p_3}$ s.t.
$$g(x_{p_1}, x_{p_2}) + g(x_{p_2}, x_{p_3}) \geq g(x_{p_1}, x_{p_3}),$$
- be symmetric so that $\forall x, y \in S$, $g(x, y) = g(y, x)$, and
- be such that $\forall x \in S$, $g(x, x) = 0$.

For example if S is the set of all towns then the distance between towns as the crow flies is a metric.

An *ultrametric*[Wei07b][1] is a metric that replaces the triangle inequality with

- $\forall x_1, x_2, x_3 \in S, \exists$ a permutation $x_{p_1}, x_{p_2}, x_{p_3}$ s.t.
$$g(x_{p_1}, x_{p_3}) \geq \min(g(x_{p_1}, x_{p_3}), g(x_{p_2}, x_{p_3}))$$

i.e. there is a tie for the minimum.

An example of an ultrametric that will have some relevance later in this Chapter is:

Theorem 1. *The depth of the m.r.c.a. of leaves in a bifurcating tree is an ultrametric.*

Proof. The symmetry property holds because the m.r.c.a. property is symmetric, i.e. x's m.r.c.a. with y is the same as y's m.r.c.a. with x.

The $g(x, x) = 0$ property is vacuous, because the m.r.c.a. of a leaf node with itself is undefined.

The strengthened triangle inequality holds because, in a bifurcating tree, for any choice of 3 leaves, either (1) two are in one subtree of the root and one is in the other, or (2) all are in the same subtree. In case (1) the m.c.r.a. of the pair is the same subtree has depth > 0 but the m.r.c.a. for pairs in different subtrees of the root have depth 0. Case (2) requires proof by induction: apply the above argument to the subtree that the leaves are in, if the root of the subtree is the m.r.c.a. then we have reduced the problem to case (1), otherwise try case (2) again (eventually (1) must work because the m.r.c.a. always exists). □

[1]Strictly the definition is that of a *min-ultrametric*, but we will make no distinction in this report between variants of the UM relationship.

2.2.1 Ultrametric matrices

The definitions in this section are based on those in [Gus97].

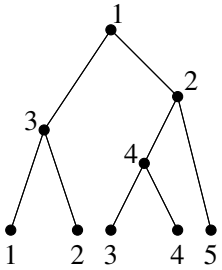
Definition 1. *Let D be a real symmetric $n \times n$ matrix. An* ultrametric tree *for D is a rooted tree T such that:*

1. *T has n leaves, each corresponding to a unique row of D;*
2. *each internal node of T has at least 2 children;*
3. *for any two leaves i and j, $D(i, j)$ is the label of the most recent common ancestor of i and j; and*
4. *along any path from the root to a leaf, the labels strictly increase.*

Such a tree and matrix are shown in Figure 2.3.

	a	b	c	d	e
a	0	3	1	1	1
b	3	0	1	1	1
c	1	1	0	4	2
d	1	1	4	0	2
e	1	1	2	2	0

(a) Symmetric matrix D



(b) Ultrametric tree T for D

Figure 2.3: Correspondence between symmetric matrix and ultrametric tree

Definition 2. *A symmetric matrix D is an* ultrametric matrix *if and only if for every set of 3 distinct indices i, j and k, there is a tie for the minimum of $D(i, j)$, $D(i, k)$ and $D(j, k)$; and $D(i, i) = 0$ for all i.*

For example, the matrix in Figure 2.3(a) is an ultrametric matrix. It should be clear that this definition means that an ultrametric matrix describes an ultrametric distance function.

There is a correspondence between ultrametric trees and ultrametric matrices that will be important in this project:

Theorem 2. *A symmetric matrix D has an ultrametric tree T if and only if it is an ultrametric matrix. Furthermore, the tree T uniquely determines the matrix D and the matrix D uniquely determines the tree T.*

Proof. Given in [Gus97]. □

The result and the proof itself have several practical repercussions:

- The uniqueness part means that trees and matrices are interchangable and we will take advantage of this property by solving the supertree problem using ultrametric matrices instead of ultrametric trees (Section 2.5).
- The proof of the $\Leftarrow$ direction is constructive and it provides an algorithm for creating an ultrametric tree from an ultrametric matrix. The existence of such an algorithm enables the use of the particular constraint modelling that was used to solve the supertree problem (Section 2.5.3).

2.3 Species trees

The theory of evolution by natural selection (see [Daw06]) is an explanation of how complex species could arise from simpler antecedents. The basic theory is that within species individuals have different characteristics (or *adaptations*) due to genetic mutation. Those adaptations that help their possessors to survive and reproduce will be present in a higher proportion in the next generation compared to unhelpful adaptations. This is by virtue of the fact that children inherit their parents' genes and hence their adaptations.

This alone is enough to explain how generations can adapt *within* species to changes in their habitat, but it remains to explain how speciation can occur[2]. Speciation occurs as a result of limited mating whereby adaptations are no longer spread approximately uniformly through the gene pool and hence certain subgroups of the species can develop certain adaptations in greater proportion. Selective mating can occur as a result of

- physical barriers whereby mating between the divided groups becomes impossible, e.g. the formation of a dividing range of mountains or the disconnection of two elevated areas of land by water;
- by sexual selection when individuals reproduce with other individuals who are in some way similar to themselves, e.g. cichlid fish females that needlessly mate only with similarly coloured males;
- human intervention, e.g. dog breeding;
- and many more. (see [Var07b] and [DW04])

These divergent subgroups will become separate species when each subgroup develops an adaptation that is individually neutral or positive, but which is negative when combined with the other group's adaptation. This is because the offspring of the two populations have become unviable, individuals that do not interbreed will be advantaged, and a gene for this behaviour will tend to dominate the gene pool as a result.

Such speciation can be drawn as a rooted bifurcating tree (a *species tree* in the biological parlance), where internal nodes represent ancestral species and leaves represent new species[3]. See Figure 2.4 for an example of this sort of tree.

A well-known example of this is the relationship between humans and the Pan group containing chimps and bonobos (see Figure 2.5). Our most recent common ancestor (m.r.c.a.) with the chimp was probably superficially more similar to chimps than to humans. It is believed that the formation of the Rift Valley in Africa is the event that heralded the divergence of the human and chimp lines. Contrary to popular belief, we are not descended *from* chimps, but rather share this now extinct m.r.c.a. with them. Chimps and bonobos are in turn commonly descended from a different m.r.c.a. which is more recent than that of chimps and humans. Humans are equally closely related to chimpanzees and bonobos.

This example demonstrates a few key features of species trees:

[2]Dawkins[DW04] defines a species by saying that two types of animals are different species when they cannot together produce offspring. Hence a species is a set of animals where each (male,female) pair could technically reproduce together. The meaning of "species" is a subject of much disagreement amongst biologists but we can avoid getting bogged down in this discussion because the intuitive meaning that they are the "same kind of animal" will suffice. For a general discussion see [DW04].

[3]Again, biologists would disagree with this because there is evidence to believe that genetic material has jumped from subtree to subtree, creating a cycle. From example from page 395 of [DW04], cell mitochondria used to be a free-living liveform, but it is now part of animal cells. This would create an edge between two different subtrees.

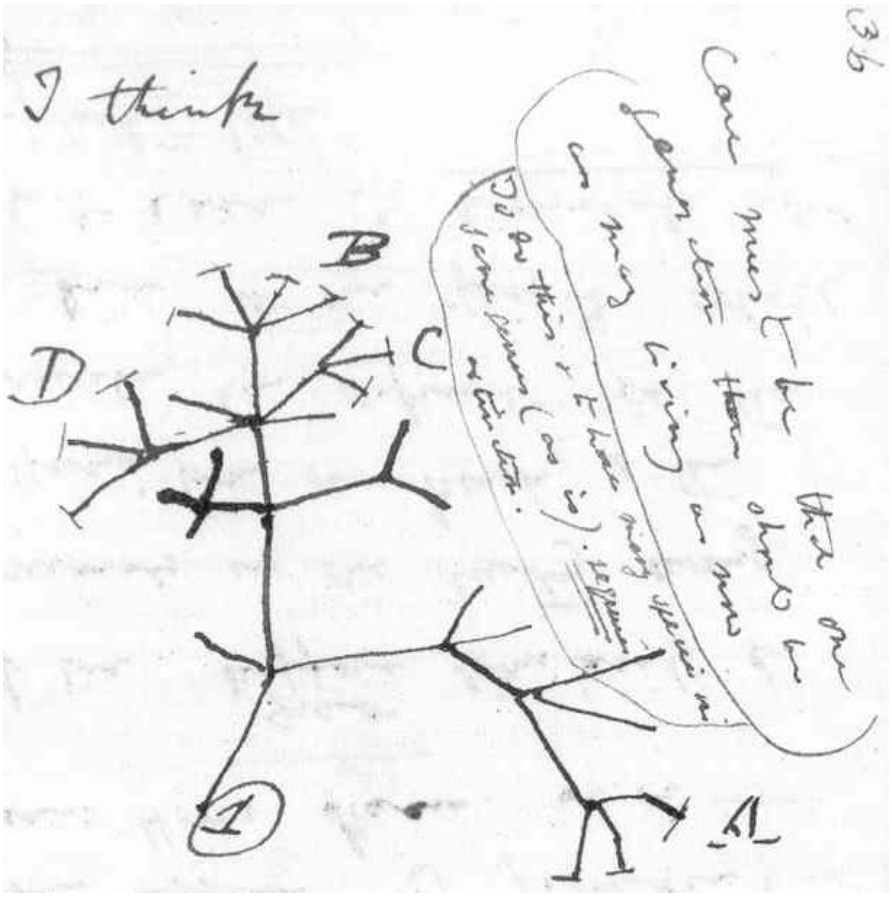


Figure 2.4: Charles Darwin's first sketch of an evolutionary tree from his *First Notebook on Transmutation of Species* (1837)

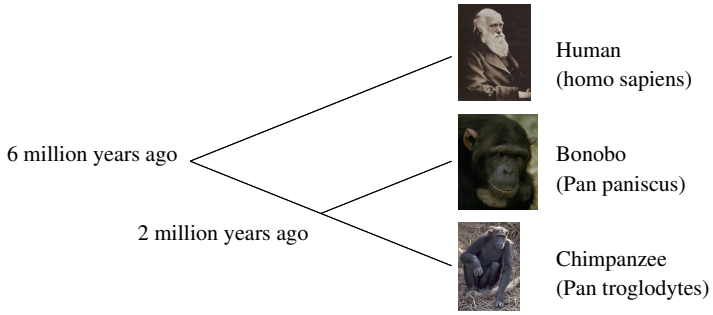


Figure 2.5: Tree of life fragment involving humans and chimpanzees. Dates taken from [DW04]. Chimp images: H. Vannoy Davis © California Academy of Sciences.

1. they are bifurcating because groups split in two at divergence events;
2. the leaves are the resultant species;
3. the internal nodes are ancestral species that are assumed to be extinct (superceded by the resultant species);
4. each pair of species has a m.r.c.a.;
5. if the internal nodes were labelled with a divergence date (e.g. years since the formation of the planet) then the labels would increase on paths from the root; and
6. a species tree need not contain every single species, but non-trivial trees contain 3 or more.

The clear correspondence between these facts and Definition 1 above demonstrate that species trees are ultrametric. Hence in solving computational problems with species trees we can exploit their dual representation as ultrametric matrices if this is worthwhile for reasons of ease of programming or efficiency.

2.4 The supertree problem

We can now discuss the immediate background to the work presented in this report: solving the supertree problem efficiently.

The supertree problem is to take a set S of species trees and to produce another species tree T that contains all the species in S and is *consistent* with every tree in S. Precisely, a tree U is consistent with the result T if and only if $U = T''$ where T'' is obtained by doing the following:

1. Let R be the set of leaves of T that are in U.
2. Obtain T' by removing leaves from T that are not in R as well as any edges incident to them.
3. To obtain T'': wherever there is a path consisting of degree 1 nodes in T', replace it with a single edge.

For examples of compatible and incompatible trees see Figures 2.6 and 2.7.

2.4.1 Applications of the supertree problem

The supertree problem has practical applications in biology where systematists attempt to build the complete tree of life. It would be too great a task for a person or group of people, and the data doesn't exist to automatically build a complete tree of life. Hence only trees involving subsets of all species are available. A solution to the supertree problem can be applied to these trees to obtain a supertree that conserves all the relationships between the species. This is most worthwhile when input trees have species in common, because it may allow new relationships to be inferred. For example in Figure 2.8, tree S says that monkeys and men are more closely related to each other than to mice, and tree T says that mice and moles are more closely related to each other than to monkeys. From this we can infer that monkeys and men are closer to each other than to moles (tree R in Figure 2.8)!

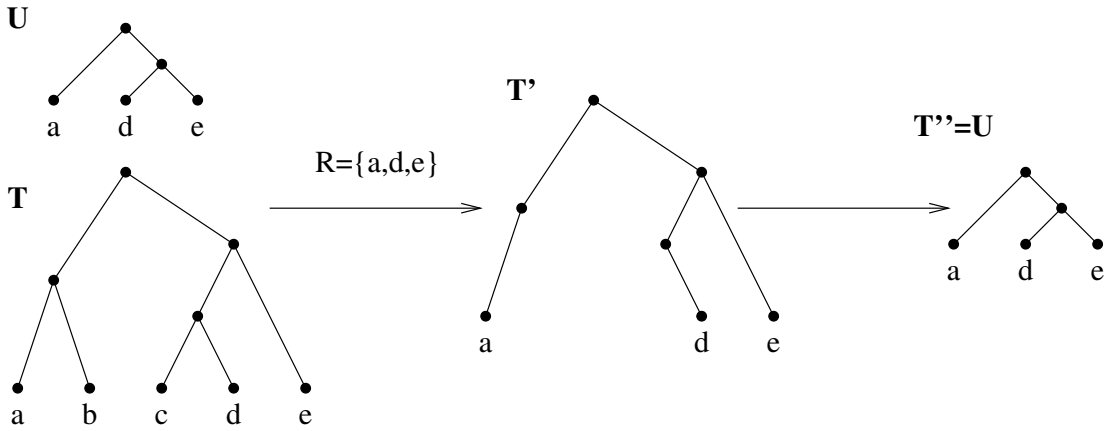


Figure 2.6: An example of a tree U compatible with a tree T

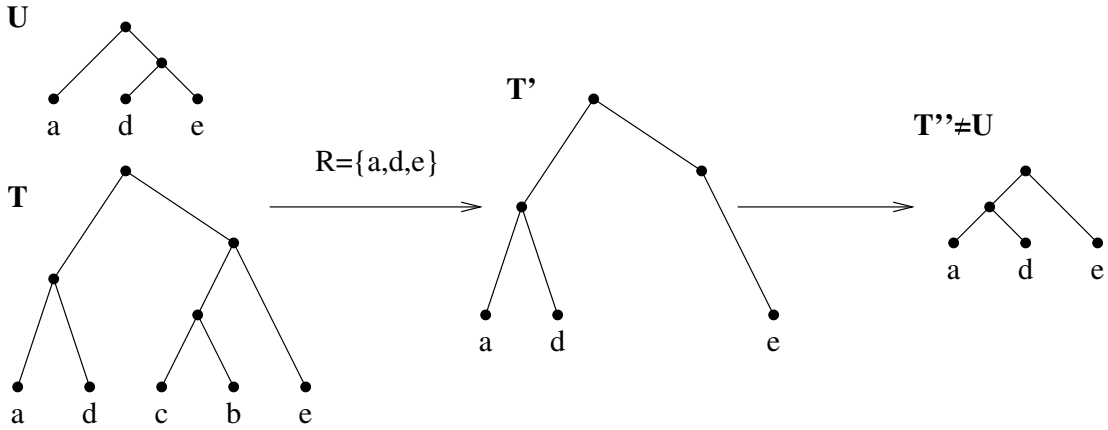


Figure 2.7: An example of a tree U not compatible with a tree T

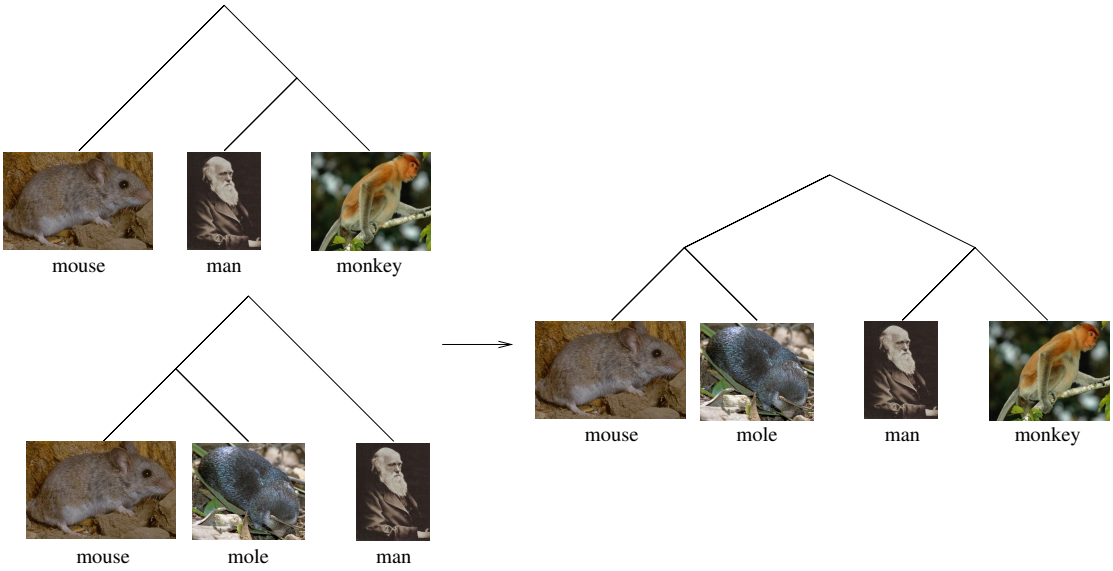


Figure 2.8: Two trees and a supertree. Monkey and mole images: © Dave Mangam. Mouse image: George W. Robinson © California Academy of Sciences

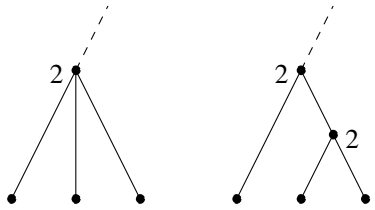


Figure 2.9: The equivalence between triples and fans

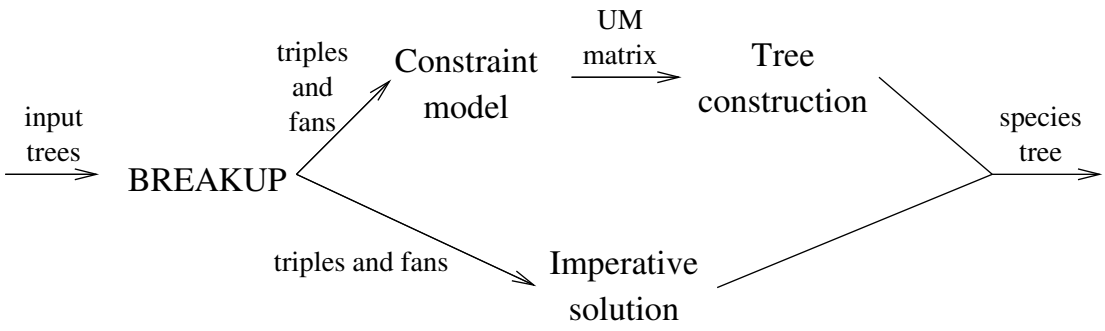


Figure 2.10: Overview of solutions to the supertree problem

2.4.2 Problems with real biological data

Methods used to derive trees may be approximate or subjective, hence different trees may contradict each other. Biologists may be able to resolve these inconsistencies[KP02] but our formulation of the supertree cannot produce a result.

Another problem is that although the tree of life ought to be strictly bifurcating, in the absence of data biologists may make the m.r.c.a. for 3 or more species be the same. This type of structure is called a *fan*. They are saying that they do not know which species diverged first and they do not want to commit. Thankfully this is equivalent to normal bifurcation and need not be treated separately. The equivalence is that a fan is indistinguishable from two connected internal nodes with the same label, see Figure 2.9 for a demonstration. However our definition of an ultrametric tree was that the labels *strictly* increase from the root and an implementation involving fans needs to remove the strictness requirement.

2.5 Solutions to the supertree problem

A variety of algorithms for solving the supertree problem have been proposed (see [NW96] for one of them). Recently in [GPSW03] and [Pro06] constraint programming techniques have been applied to the problem with some success. In this section we will describe some previous work in these areas and identify the specific problem this project addresses.

Briefly, the approach taken in the above cited solutions is to break input trees up into a minimal set of ultrametric trees of size 3 that describe them completely, and then to find one or more trees compatible with all of these. For imperative and constraint programming solutions the first stage is identical but the second stage is quite radically different. See Figure 2.10 for an overview.

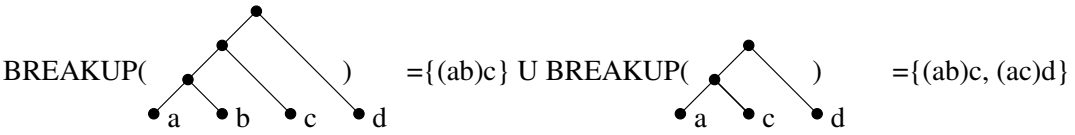


Figure 2.11: The BreakUp algorithm

2.5.1 1st stage—breaking up the input trees

The BreakUp algorithm presented in [NW96] takes as input a species tree and returns a set of *triples* and *fans* that completely describe the tree. A triple $(bc)a$ says that species b and c are more closely related to each other than either is to a. A fan abc says that a, b and c should be pairwise equally closely related. An understanding of evolution suggests that fans should be impossible, however their purpose is not to assert the relationship actually exists but to represent a lack of knowledge about what the true relationship is (see Section 2.4.2). Hence abc can be read as "either $(ab)c$, $(ac)b$ or $(bc)a$".

If $m(x, y)$ is the m.r.c.a. function for the input tree then $(ab)c$ means $m(a, b) > m(a, c) = m(b, c)$.

Details of the BreakUp algorithm are not important for this project, but for an example execution see Figure 2.11.

Intuitively, the set of triples and fans produced by the BreakUp algorithm describe relationships between certain subsets of the nodes, but it is possible to derive any relationship that existed in the input tree by inference. This corresponds to an intuitive understanding that the BreakUp algorithm retains all the information in the tree but uses an alternative representation. In Figure 2.11 the relationship $(bc)d$ does not appear in the output but it can be inferred from the other triples as follows:

From $(ab)c$ we have $m(a, b) > m(a, c) = m(b, c)$ and from $(ac)d$ we have $m(a, c) > m(a, d) = m(c, d)$. Hence $m(b, c) = m(a, c)$ and $m(a, c) > m(c, d)$ and so $m(b, c) > m(c, d)$.

From Theorem 1 we know that the m.r.c.a. function is ultrametric and so either $m(b, c)$, $m(b, d)$ and $m(c, d)$ are all the same or two are the same and one is greater. Since $m(b, c) > m(c, d)$ are different we can infer that $m(b, c)$ and $m(b, d)$ must be the same, i.e. $m(c, d) = m(b, d)$.

Hence $m(b, c) > m(c, d) = m(b, d)$ and we have derived $(bc)d$ as required.

We will now formalise this notion that the triples and fans describe the tree:

Theorem 3. *The input tree T to the* BreakUp *algorithm is the unique tree compatible with all of the triples produced by* BreakUp*.*

Proof. Given in [NW96]. □

Corollary 1. *Any relationship in the input tree T to* BreakUp *must be able to be inferred from the result triples.*

Proof. By equivalence of $(ab)c$ and $m(a, b) > m(a, c) = m(b, c)$, and Theorem 3. □

2.5.2 2nd stage—solving the supertree problem

[NW96] describes an imperative algorithm using BreakUp as a preprocessing step. This runs in around $O(n^3)$ time where n = the number of species in the tree[4]. This algorithm can find either one solution or all solutions.

[4]The complexity proved in the paper is tighter than this, but here it is simplified because only the fact that it is polynomial and the rough exponent matters for our purposes.

Two slightly different constraint models have been proposed to solve slightly different problems:

2.5.2.1 The [GPSW03] model

This model is intended to solve the basic supertree problem of taking a set of input trees and constructing one or more trees consistent with all of them.

The variables are a symmetric $n \times n$ matrix M of integer variables with domains $1, \ldots, n-1$ or 0 on the main diagonal. Variable $M(i,j)$ is the depth of the m.r.c.a. of species i and j. The reason for the $1, \ldots, n-1$ domains is that these allow a distinct label to be given to every layer containing leaf nodes in a tree with the maximum depth of $n-1$.

First, constraints are posted to make the whole matrix ultrametric, thus ensuring that any resulting species tree is ultrametric:

$$\begin{array}{ll} & M(i,j) > M(i,k) = M(j,k) \\ \vee & M(i,k) > M(i,j) = M(j,k) \\ \vee & M(j,k) > M(i,j) = M(i,k) \\ \vee & M(i,j) = M(i,k) = M(j,k) \end{array} \tag{2.1}$$

for each $i < j < k$. Furthermore for each triple $(ij)k$ the constraint

$$M(i,j) > M(i,k) = M(j,k) \tag{2.2}$$

is posted and for each fan ijk

$$M(i,j) = M(i,k) = M(j,k) \tag{2.3}$$

is posted. These constraints break the disjunctions of 2.1 to allow a meaningful solution.

The model has $\frac{n^2-n}{2}$ variables and

$$t + f + C(n,3) = t + f + \frac{n(n-1)(n-2)}{6} = O(n^3) + O(n^3) + O(n^3) = O(n^3) \tag{2.4}$$

constraints, where t is the number of triples and f the number of fans. There are $O(n^3)$ of each because each one breaks the disjunction in at most one constraint from Equation 2.1, and there are $O(n^3)$ of them.

2.5.2.2 The [Pro06] model

This model sets out to solve a slightly different problem where instead of the M variables representing a depth in the tree, they represent a possible range of divergence dates for that internal node. This means that using the M variables as search variables would not work because we do not want them to be instantiated. Instead they should contain a possible range of values. For this reason an auxiliary $D(i,j,k)$ variable is added for each $i < j < k$, each has a domain of $1, 2, 3, 4$ and represents the relationship choice made for the 3 species. 1 means that i and j are most closely related, 2 that i and k are closest, 3 that j and k are closest and 4 that all 3 are equally close. This is encoded by replacing the constraints in Equation 2.1 with

$$\begin{array}{ll} & D(i,j,k) = 1 \wedge M(i,j) > M(i,k) = M(j,k) \\ \vee & D(i,j,k) = 2 \wedge M(i,k) > M(i,j) = M(j,k) \\ \vee & D(i,j,k) = 3 \wedge M(j,k) > M(i,j) = M(i,k) \\ \vee & D(i,j,k) = 4 \wedge M(i,j) = M(i,k) = M(j,k) \end{array} \tag{2.5}$$

and making D be the search variables.

Another difference is that instead of posting the constraints in Equations 2.2 and 2.3 the triples and fans can be posted by instantiating a D variables, e.g. for fan $(ij)k$ set $D(i, j, k)$ to 1.

The number of ultrametric constraints in this model is still $O(n^3)$, but each is more complex. All the constraints for triples and fans have been removed.

2.5.3 3rd stage—Producing a tree

Using the constructive proof of the $\Leftarrow$ direction of Theorem 2 a tree can be produced from the ultrametric matrix of variables and this tree is the final result.

2.5.4 Advantages and disadvantages of a CSP encoding and the aims of this project

The main motivation for a CSP solution to a problem is that models are comparatively easy to derive and to change, whereas imperative solutions can be very resistant to the addition of side constraints. A case in point is that the addition of ancestral divergence dates which is almost trivial in CP would require a complete rethink of an imperative solution. It also becomes easier to optimise solutions by a scoring function, perhaps to weight tree evidence according to biologists' confidence in it and so produce the tree with the greatest overall weight of evidence.

Some drawbacks with the constraint solution to the supertree problem is that the worst case time complexity may be as poor as $O(n^{n^2})$ and the space complexity is $O(n^3)$ with a large constant factor. The practical repercussion of this is that modest sized species data cannot be loaded into the memory of a typical workstation, and that the algorithm from [NW96] can solve problems in reasonable time that the constraint model cannot. Later in this report we will show that for both constraint models the worst case space complexity has been improved to $O(n^2)$. Furthermore, a preliminary result regarding the time complexity of the first model has improved it to $O(n^4)$. Another possible improvement to the specific modelling presented in [Pro06] is that it may be possible to maintain a higher level of consistency in the ultrametric constraints, thus reducing the size of the search space.

This project is a contribution towards solving these problems of time complexity and space complexity and a contribution towards a better understanding of the constraint model as a whole. This was achieved by designing and implementing custom constraints for Equations 2.1 and 2.5 doing generalised bounds arc consistency. Next they were adapted to occupy constant space. Finally a comparative study of these different constraints to the benchmark implementation was carried out.

Chapter 3

3 variable constraints

"After three leaps, even a toad needs a rest."— **Chinese proverb**

In this section several designs of 3 variable ultrametric constraints will be presented, proved correct and implemented.

3.1 Introduction

The ultrametric constraint on 3 variables (henceforth, UM-3) constrains 3 variables x, y and z as follows:

$$\begin{array}{ll} & x > y = z \\ \vee & y > x = z \\ \vee & z > x = y \\ \vee & x = y = z \end{array} \tag{3.1}$$

The intuition to bear in mind for this is that it makes sure that there is a tie for the least element of the three, i.e. either all three are the same, or two are the same and the other is greater.

A pre-existing UM-3 constraint from work by Prosser and others (see [GPSW03] and [Pro06]) implemented in JCHOCO is the baseline against which the new constraints presented in this chapter are to be measured. It is described in Section 3.3 to set the scene. The problem with the constraint is that it is implemented as a compound constraint involving 23 others[1], and uses approximately 23 times the space that a single UM-3 constraint would. The huge reduction is memory usage that would result from use of a custom constraint would allow *larger* species problems to be loaded and in addition it might achieve the same or better propagation in less time, meaning problems could be solved *quicker*.

In addition to this improvement, the particular circumstance where this constraint is to be used in the species model can be exploited by the use of a whole matrix constraint. Now the whole matrix could be constrained to be UM in constant space and so the memory occupied by the whole model would be improved from $O(n^3)$ to $O(n^2)$, since it is now dominated by the constant space required for each variable in the matrix, rather than the constraints on it. The matrix constraint may also run faster than the custom constraint and will maintain the same level of consistency.

Finally we will see an experimental forward checking constraint that was originally intended as a straw man but ended up being surprisingly effective in practice, as will be shown in Sections 5.1.3 and 5.2.2.7.

[1] Count the occurences of `pb.and`, `pb.eq`, `pb.gt` and `ty.or` in Listing 3.1 to see how many constraints are involved in the whole thing.

```
1  //Put together a few built-in constraints to do ultrametric constraint
2  //over 3 variables.
3  public class ToolkitUm3 {
4      public static Constraint makeUltrametric(IntDomainVar x, IntDomainVar y,
5                                                IntDomainVar z) {
6          AbstractProblem pb = x.getProblem();
7          Constraint U1 = pb.and(pb.eq(y,z),pb.and(pb.gt(x,y),pb.gt(x,z)));
8          Constraint U2 = pb.and(pb.eq(x,z),pb.and(pb.gt(y,x),pb.gt(y,z)));
9          Constraint U3 = pb.and(pb.eq(x,y),pb.and(pb.gt(z,x),pb.gt(z,y)));
10         Constraint U4 = pb.and(pb.eq(x,y),pb.and(pb.eq(x,z),pb.eq(y,z)));
11         Constraint U5 = pb.or(U1,pb.or(U2,pb.or(U3,U4)));
12         return U5;
13     }
14 }
```

Listing 3.1: Implementation of TOOLKIT-UM-3

3.2 Expected and intended behaviour of the UM-3 constraint

Theorem 4. *When each variable has the same domain* $1, \ldots, n$ *then the number of distinct satisfying assignments for* UM-3 *is* $\frac{1}{2}n(3n-1)$.

Proof. Either all the values in the assignment are the same, or the largest value is in one of three positions and the other two must be smaller. In the former case there are n assignments, one for each value in the domain of the variables. In each of the latter cases, the larger value can be assigned $n, n-1, \ldots, 2$ and these leave respectively $n-1, n-2, \ldots, 1$ ways to assign the remaining two variables identically. Hence by the sum rule[Ros98] there are $(n-1)+(n-2)+\ldots+1 = \frac{1}{2}n(n-1)$ ways to assign in each of these cases. Hence, by another application of the sum rule, overall there are $n + 3\frac{1}{2}n(n-1) = \frac{1}{2}n(3n-1)$ assignments as required. □

This fact suggests an informal way to test implementations of the UM-3 constraint, by ensuring that it returns the correct number of solutions when posted over 3 variables with domains $1, \ldots, n$. Furthermore this equation gives a quantitative measure of the tightness of the UM-3 constraint because the number of correct instantiations is $\frac{1}{2}n(3n-1)$ out of n^3 in total.

More confidence in an implementation can be provided by proving the theoretical correctness of a constraint. This involves proving that no values are removed from domains during propagation that could be part of a solution. All the constraints in this chapter are either evidently correct, or a proof is provided. In some cases proofs will be given that a constraint maintains a certain level of consistency, for example to prove that a binary constraint maintains AC we must prove that after propagation any and all values in both domains are supported by at least one value in the other domain. This notion generalises in an obvious way to other levels of consistency.

3.3 The TOOLKIT-UM-3 constraint

The toolkit constraint uses inbuilt constraints provided by JCHOCO. It is trivially easy to create and the general idea is guaranteed to be correct since the definition of the constraint in code is almost identical to the mathematical definition. There remains a slight possibility that a particular solver will be incomplete or unsound, but if this was the case no constraint implementation could ever be guaranteed to be correct!

The implementation in Listing 3.1 is a factory for UM constraints, the resulting constraint should be posted by the caller. There is a clear equivalence between this

implementation and Equation 3.1, except that in each conjunction the 3rd constraint is redundant[2].

3.4 The CUSTOM-UM-3 constraint

In this section, we will describe the design of a UM-3 constraint maintaining generalised bounds arc consistency (GBAC). GBAC is a combination of BAC and GAC; what it means for an n-ary constraint is that the upper and lower bounds of each variable must be involved in a satisfying assignment with arbitrary values in all of the other $n-1$ variables, i.e. they must be *supported*. There remains the possibility that a value in the middle of a domain could be unsupported.

3.4.1 Analysis of cases for lower and upper bounds

Once propagation is complete each bound b must be supported, either

- by v the same as it and w greater than it ($b = v < w$), or
- by v and w the same as it ($b = v = w$), or
- by v and w smaller than it ($b > v = w$).

Since this is a very important notion and to recapitulate, this means that every bound must be associated with a value in each of the other two domains so that there is a tie for the minimum among the 3 values. An unsupported bound must be removed from the domain and a supported bound must not be removed. Due to the properties of the search algorithms propagation is not done when any domain is empty (see Section 3.4.2.1).

We will first present the procedure LBFIX (Listing 3.2) used to make sure all lower bounds (l.b.s) are supported, then show why it works. The intuitive understanding of the procedure is that when an l.b. is strictly less than the other two, it cannot be supported because in UM there is always a tie for the minimum value in 3. By removing this overhang the l.b.s support each other and the job is done.

```
1  let s be the smallest lower bound
2  let m be the middle lower bound
3  let l be the largest lower bound
4  if(s is less than both m and l)
5     s:= m
```

Listing 3.2: Pseudocode for lower bound propagation procedure LBFIX for CUSTOM-UM-3

The possible states of lower bounds when LBFIX is applied can be divided into 13 cases (Figure 3.2), some of which are symmetric (1-6 and 8-10). The reason why these cover all possibilites is that either all the l.b.s are different (1-6), or two are the same and one different (8-13), or all three are the same (7); and that these possibilities can be reordered over the 3 variables.

These cases show the relationships between the lower bounds at a point in time when some bounds may be unsupported. What these diagrams are *not* supposed to suggest is that, for example in case 1, the bounds differ by 1. Rather when two bounds are lined up they are the same bound and when one is different from another they are different by some non-zero but unspecified amount. Hence they describe the *relationship* and not actual values. This intuition is fleshed out in Figure 3.1.

[2]For reasons unknown, the constraint maintains a lower level of consistency without the redundant constraints. This may not be the case in other constraint toolkits and this issue underlines how unpredictable the behaviour of constraint solvers can be.

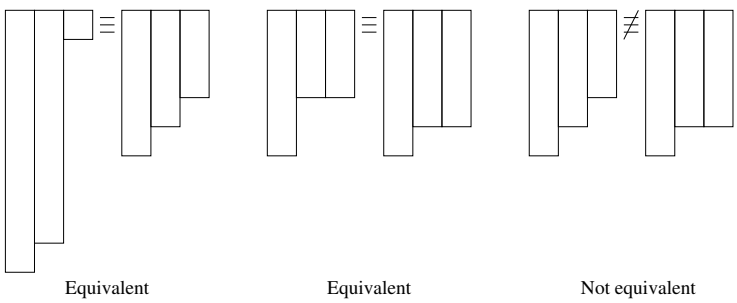


Figure 3.1: The intuition for understanding box diagrams

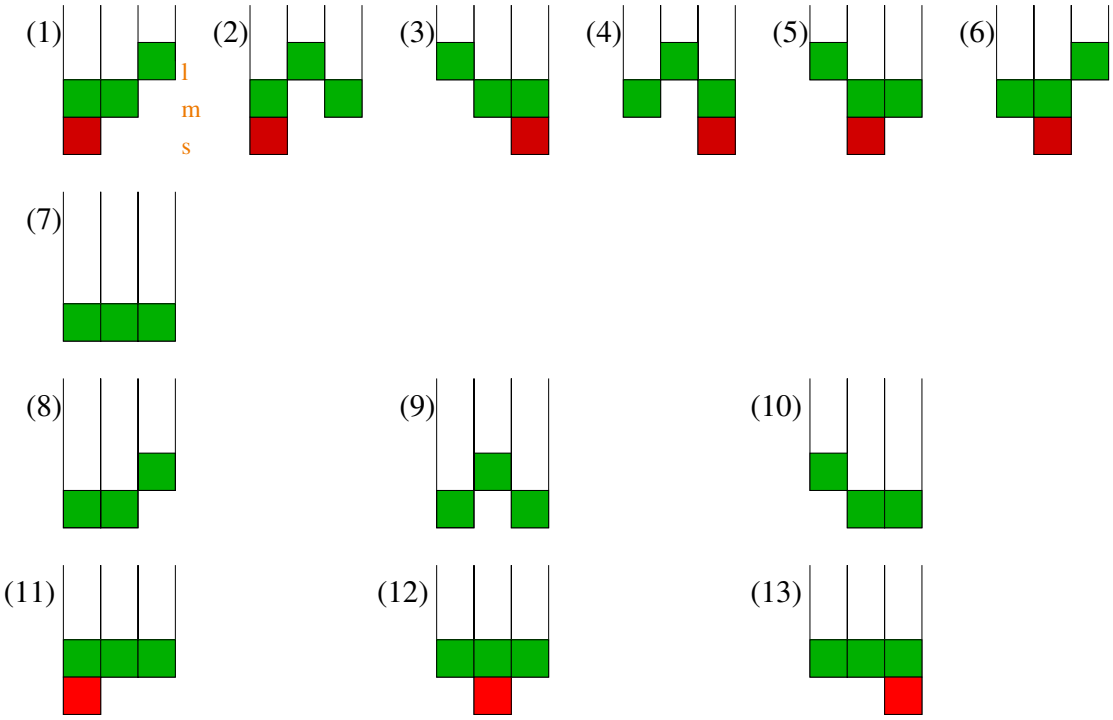


Figure 3.2: Cases for analysis of lower bounds in UM-3

Lemma 1. *The changes made to lower bounds by* LBFIX *remove all unsupported values and no supported values are removed.*

Proof. In cases (1)-(6) the lower bounds are distinct, the red shaded range is unsupported. LBFIX sets $s \leftarrow m$ and so the unsupported range is removed. The new l.b.s shaded in green are mutually supportive in the $u > v = w$ configuration. In case (7) each l.b. is supported in the $u = v = w$ configuration and LBFIX makes no changes. In cases (8)-(10) each is supported in the $u > v = w$ configuration and no changes are made. Finally, in cases (11)-(13) the red shaded range is unsupported but LBFIX's effect is to remove it and the new lower bounds shaded in green are mutually supportive. □

The corresponding procedure UBFIX for upper bounds is only slightly more complex (Figure 3.3):

```
1  let s, m and l be the smallest, middle and largest u.b.s respectively
2                                              (breaking ties arbitrarily)
3  let S, M and L be the domains containing s, m and l respectively
4  if(s is less than both m and l)
5     if(S and L have null intersection)
6        m:= s;
7     else if(S and M have null intersection)
```

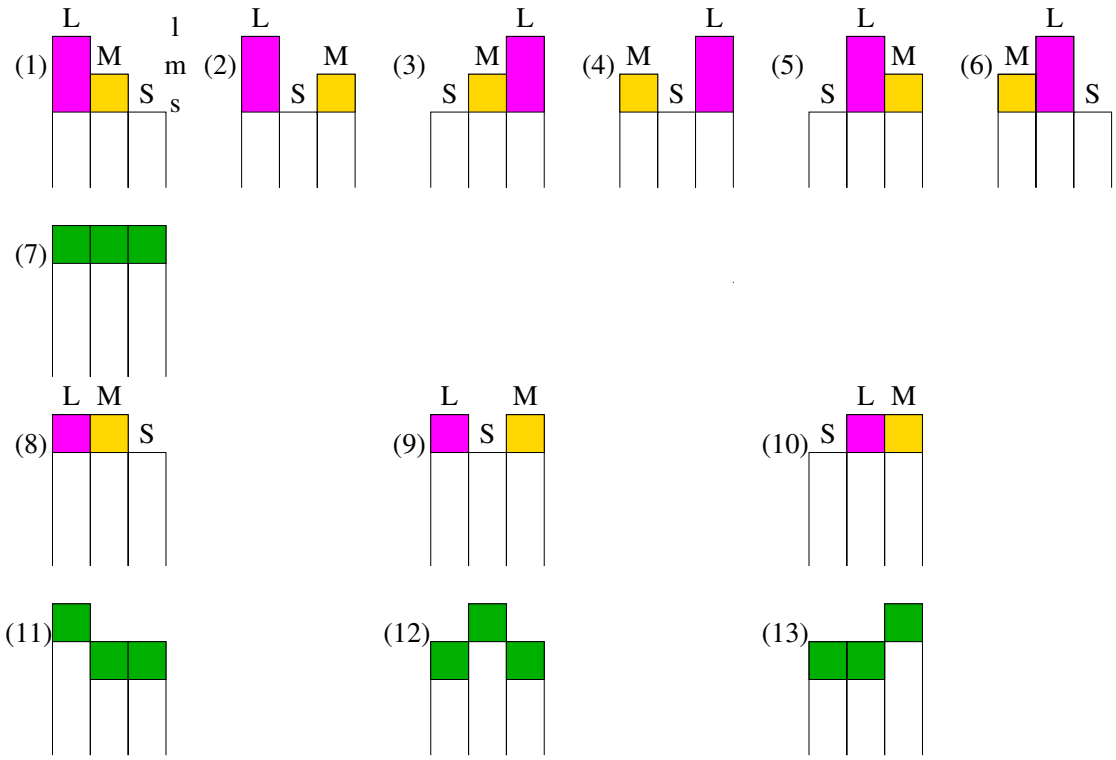


Figure 3.3: Cases for analysis of upper bounds in UM-3

```
8        l:= s;
```

Listing 3.3: Pseudocode for upper bound propagation procedure UBFIX for CUSTOM-UM-3

The possible starting states for UBFIX can be summarised by the 13 cases shown in Figure 3.3.

Lemma 2. *The changes made to upper bounds in carrying out* UBFIX *remove all unsupported values and no supported values are removed.*

Proof. Let S, M and L be the domains with least, middle and greatest upper bounds, breaking ties arbitrarily. Let s, m and l be the upper bounds of S, M and L respectively.

Cases (7) and (11)-(13): the upper bounds are mutually supportive, and no changes result from the procedure.

Cases (1)-(6): If the gold shaded portion of the domain M is unsupported then line 5 of UBFIX will trim it. The new upper bounds l, s and s of L, M and S are mutually supportive.

If the gold shaded portion of M is supported and the magenta shaded portion of L is not supported then it is trimmed by line 7 of UBFIX. The new upper bounds s, m and s of L, M and S are mutually supportive.

If the gold shaded portion of M and the magenta shaded portion of L are both supported then $S \cap L \neq \phi$ and $S \cap M \neq \phi$ and so no changes result from UBFIX.

Cases (8)-(10): Suppose w.l.o.g. that L is the domain shaded magenta and M the domain shaded gold.

If $S \cap L = \phi$ and $S \cap M = \phi$ then the first conditional branch at Line 4 is executed. This results in line 5 removing the whole of domain M and this is correct because

no values in M or L equal any value in S and so there can be no tie of minimum and no possible instantiation.

If $S \cap L = \phi$ then the magenta region can only ever be the maximum of three. Hence the gold region is unsupported and is removed. This change means the u.b.s are mutually supportive.

For $S \cap M = \phi$, the argument is symmetric.

□

3.4.2 The propagation algorithm

Having proved these two Lemmas 1 and 2 about LBFix and UBFix the entire propagation algorithm (Listing 3.4) can be presented:

```
1 if(receive notification of change in lower bound)
2    do LBFix
3    do UBFix
4 else if(receive notification of change in upper bound)
5    do UBFix
```

Listing 3.4: Pseudocode for Custom-Um-3 propagation

This works because a change to an upper bound can only affect the support for other upper bounds, but a change to a lower bound can affect the support for both lower and upper bounds. It would be correct to cycle between fixing lower and upper bounds repeatedly until all were supported, but we can save the effort of checking by being careful only to do the minimum necessary work.

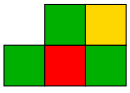
Lemma 3. *It is possible for a change in a lower bound to result in the loss of support for another lower bound.*

Proof. All the bounds in the diagram have support, but when the red shaded l.b. is lost the gold shaded l.b. loses support.

□

Lemma 4. *It is possible for a change in a lower bound to result in the loss of support for an upper bound.*

Proof. All the bounds in the diagram have support, but when the red shaded l.b. is lost the gold shaded u.b. loses support.

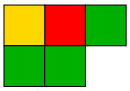


□

The existence proof of Lemma 4 does not adequately capture *why* this should be the case. The reason is that upper bounds may not be mutually supportive, meaning that the loss of l.b. may remove support for a u.b. The opposite of the Lemma is not true, because, as we will prove in Corollary 2, a change in an upper bound cannot possibly cause the loss of support for any lower bound.

Lemma 5. *It is possible for the loss of an upper bound to cause the loss of support for another upper bound.*

Proof. All the bounds in the diagram have support, but when the red shaded u.b. is lost the gold shaded u.b. loses support.



□

Lemma 6. *When lower bounds are supported, they support each other.*

Proof. Consider 3 supported lower bounds. Suppose for a contradiction that the two least of these are distinct. One of these is distinct lowest and it cannot be supported on account of the fact that it is not equal to anything or larger than anything. Therefore by contradiction the two least must be equal. However the other l.b. is at least as large as these, so the l.b.s are mutually supportive. □

As well as having an elegant little proof, this property of the algorithm will be of key importance in later work, when we suggest that with the addition of the new constraint having this property, AC is now sufficient to solve the supertree problem without search.

Corollary 2. *It is impossible for a change in an upper bound to result in the loss of support for a lower bound.*

Proof. By Lemma 6 an l.b. retains support as long as the other l.b.s are intact, hence losing an upper bound has no effect. □

This Corollary suggests that a further improvement on the algorithm in Listing 3.4 is to execute Line 3 if and only if the l.b. lost is the only remaining support for a u.b. However, the conditionals intrinsic in UBFix amount to much the same thing and there is little point in repeating them.

The point of all these proofs has been to build up a complete proof of correctness and proof of GBAC status for the Custom-Um-3 constraint:

Theorem 5. *The procedure in Listing 3.4 does* GBAC *propagation for the* Custom-Um-3 *constraint.*

Proof. The first thing to establish is that no values are removed from domains during propagation that could be part of a solution, this is immediate from Lemmas 1 and 2 because any values are removed as a result of the corresponding procedures.

The next thing to establish is that the lower bounds are made as big as possible and the upper bounds as small as possible. In the case of the lower bounds, by Lemma 3 and Corollary 2 we know that only the loss of a lower bound can result in the need to change a lower bound during propagation. Also, by Lemma 1 the lower bound procedure makes the lower bounds as large as possible. Hence the algorithm in Listing 3.4 propagates in all the necessary cases and does the largest possible amount of propagation, as required.

In the case of upper bounds, by Lemmas 4 and 5 the loss of either a lower or upper bound can result in the need to change an upper bound. By Lemma 2 the upper bound procedure makes the upper bounds as small as possible. Hence the propagation algorithm in Listing 3.4 propagates in all the necessary cases and does the largest possible amount of propagation, as required. □

This propagation algorithm runs in $O(1)$ time.

3.4.2.1 A note on domain wipeouts

If at any point in propagation a domain empties, propagation will stop and the search algorithm will backtrack. Domain wipeouts complicate matters because it might seem as though l.b.s and u.b.s are changing simultaneously, but in fact the propagation algorithm will never be invoked and so it doesn't matter.

3.4.3 Implementation

A JAVA implementation of the constraint conformant with the JCHOCO library is shown in Listing 3.5.

Line 1 The constraint indirectly implements the `AbstractIntConstraint` interface, so it can be posted into a JCHOCO model.

Lines 2-7 An object of this class represents a sequence of 3 variables, which will be sorted by lower or upper bound.

Lines 8-10 Notify the parent class of the variables the constraint is posted over, this is how the new constraint arranges to receive propagation events.

Lines 11-14 Implement fragment of propagation algorithm in Listing 3.4 that does l.b. propagation.

Lines 15-19 Implement the rest of Listing 3.4 to do u.b. propagation.

Lines 20-24 Before search commences, arc consistency is established and in the process this method will be invoked. First fix the lower bounds and then the upper bounds. Since change in u.b.s can't affect support for l.b.s this is sufficient.

Lines 25-29 Instantiation is when a domain is reduced to a single value and it is like both the lower and upper bounds changing. It suffices to check both bounds in a safe order.

Lines 30-36 Ignore loss of non-bound values.

Lines 37-44 Straightforward implementation of LBFIX.

Lines 45-56 Straightforward implementation of UBFIX.

The utility function `nullIntersection()` in Listing 3.6 returns true if and only if the two variables it is supplied with have non-overlapping domains. It works by checking whether the lower bound of either domain is strictly greater than the upper bound of the other.

The function `sortOnInf()` in Listing 3.7 takes 3 variables and returns them in sorted order by lower bound. It uses a fixed comparison sorting procedure for 3 elements that uses at most 3 comparisons for any input. This procedure is optimal in terms of the worst case number of comparisons, because if there was to be a procedure that used at most 2 comparisons the "fattest" decision tree with depth 2 would have only 4 leaves and hence could not produce $3! = 6$ different results. Since this procedure is run at least once per propagation it pays to make it as fast as possible.

```
1  public class CustomUm3 extends AbstractTernIntConstraint {
2      private class Triple {
3          IntDomainVar s, m, l;
4          public Triple(IntDomainVar s, IntDomainVar m, IntDomainVar l) {
5              this.s = s; this.m = m; this.l = l;
6          }
7      }
8      public CustomUm3(IntDomainVar v0, IntDomainVar v1, IntDomainVar v2) {
9          super(v0, v1, v2);
10     }
11     //called whenever an inf is changed
12     public void awakeOnInf(int idx) throws ContradictionException {
13         fixInfs();
14     }
15     //called whenever a sup is changed
16     public void awakeOnSup(int idx) throws ContradictionException {
17         fixInfs();
18         fixSups();
19     }
20     //initial propagation, just check uppers and lowers
21     public void awake() throws ContradictionException {
22         fixInfs();
23         fixSups();
24     }
25     //when variable index <idx> is instantiated, just check both bounds
26     public void awakeOnInst(int idx) throws ContradictionException {
27         fixInfs();
28         fixSups();
29     }
30     public void awakeOnRem(int idx, int x) throws ContradictionException {
31         ;
32     }
33     public void awakeOnRemovals(int idx, IntIterator deltaDom)
34         throws ContradictionException {
35         ;
36     }
37     private void fixInfs() throws ContradictionException {
38         Triple si = sortOnInf();
39         int sInf = si.s.getInf();
40         int mInf = si.m.getInf();
41         int lInf = si.l.getInf();
42         if(sInf != mInf && mInf != lInf)
43             si.s.setInf(mInf);
44     }
45     private void fixSups() throws ContradictionException {
46         Triple ss = sortOnSup();
47         int sSup = ss.s.getSup();
48         int mSup = ss.m.getSup();
49         int lSup = ss.l.getSup();
50         if(sSup != mSup) {
51             if(nullIntersection(ss.l, ss.s))
52                 ss.m.setSup(sSup);
53             else if(nullIntersection(ss.m, ss.s))
54                 ss.l.setSup(sSup);
55         }
56     }
57 }
```

Listing 3.5: Implementation of CUSTOM-UM-3

```
1 //return true if and only if v and w's domains have null intersection
2 private static boolean nullIntersection(IntDomainVar v, IntDomainVar w) {
3     return v.getInf() > w.getSup() || v.getSup() < w.getInf();
4 }
```

Listing 3.6: A procedure to discover whether two domains have a null intersection

```
1  private Triple sortOnInf() {
2      int v0Inf = v0.getInf();
3      int v1Inf = v1.getInf();
4      int v2Inf = v2.getInf();
5      if(v0Inf <= v1Inf)
6          if(v1Inf <= v2Inf)
7              return new Triple(v0, v1, v2);
8          else
9              if(v0Inf <= v2Inf)
10                 return new Triple(v0, v2, v1);
11             else
12                 return new Triple(v2, v0, v1);
13     else
14         if(v0Inf <= v2Inf)
15             return new Triple(v1, v0, v2);
16         else
17             if(v1Inf <= v2Inf)
18                 return new Triple(v1, v2, v0);
19             else
20                 return new Triple(v2, v1, v0);
21 }
22
23 private Triple sortOnSup() {
24     //symmetric with sortOnInf()
25 }
```

Listing 3.7: Implementation of optimal comparison sorting procedure for 3 items

3.5 The MATRIX-UM-3 constraint

In the species model, the constraint UM-3$(M(i,j), M(i,k), M(j,k))$ is posted over the matrix M for indices $i < j < k$. This regular pattern can be exploited by a whole matrix UM-3 constraint which we will call MATRIX-UM-3. On the receipt of a propagation event for a variable $M(i,j)$, it will do normal UM-3 propagation on variables $M(i,j)$, $M(i,k)$ and $M(j,k)$ for all indices $1 \leq k \leq n$ s.t. $k \neq i$ and $k \neq j$, i.e. exactly the same triples that would previously have had an explicit constraint over them.

For example if $n = 4$ and a change occurs in $M(1,2)$ then UM-3 propagation is done for

M(1,2), M(1,3) and M(2,3) and
M(1,2), M(1,4) and M(2,4)

but *not* for

M(1,2), M(1,1) and M(2,1) and
M(1,2), M(1,2) and M(2,2)

The reason for the improvement in memory usage is that an explicitly stored extensional representation of which constraints have been posted using $O(n^3)$ space is being replace by a dynamically (and efficiently) generated intensional representation using code occupying $O(1)$ space.

The constraint is effectively mimicking part of the AC-3 algorithm since it:

1. Receives a propagation event on a variable.
2. Identifies which constraints are over that variable.
3. Arranges for the necessary propagation to be done.

If the constraint makes any changes to domains then these changes are propagated by JCHOCO's underlying AC algorithm.

This propagation algorithm runs in $O(n)$ time, since it may need to revise $n-2$ domains and each revision runs in $O(1)$ time, as was stated in Section 3.5.

3.5.1 Expected behaviour of MATRIX-UM-3

From [Fel78], the number of bifurcating trees with n leaves is

$$\frac{(2n-3)!}{2^{n-2}(n-2)!} \tag{3.2}$$

Provided that only a single UM matrix corresponding to each "shape" of tree is found (see Figure 5.1(d) for an example of different UM trees with the same shape), the number of solutions to MATRIX-UM-3 posted over a $n \times n$ matrix of $\{1, \ldots, n-1\}$ variables is given by Equation 3.2. However the standard model does not remove this symmetry and further constraints would be required to do this.

An alternative way to test the constaint is by comparing results for a single MATRIX-UM-3 constraint with result for a matrix-full of any other UM-3 constraints; they should get exactly the same solutions.

3.5.2 Implementation

The implementation of the CUSTOM-UM-3 (Listing 3.5) makes use of variables `v0`, `v1` and `v2` which tell it which variables it is posted over; these are its sole state. Hence as an implementation trick, we can copy the body of class `CustomUm3` verbatim into the body of a matrix constraint class and then manipulate these variables to convince the old `CustomUm3` code that it's doing the same job it always did.In fact, it is responsible for propagating several of these old-style constraints! In detail, if the matrix constraint wants to propagate changes to $M(i,j)$ it will set `v0` $\leftarrow M(i,j)$, `v1` $\leftarrow M(i,k)$ and `v2` $\leftarrow M(j,k)$ for each k in turn and call `fixLowers()` and `fixUppers()` to propagate into neighbouring variables each time.

This key part of the implementation is reproduced in Listing 3.8. `doPropagate()` is invoked whenever the constraint receives a propagation event affecting lower or upper bounds. See Listing C.1 in Appendix C for a complete code listing of MATRIX-UM-3.

```
1  private void doPropagate(int idx) throws ContradictionException {
2      v0 = getIntVar(idx);
3      int[] index = varToIndex.get(v0); //index of v0 in M
4      int i = index[0]; int j = index[1];
5      for(int k = 0; k < n; k++) {
6          if(i != k && j != k) {
7              v1 = mat[i][k];
8              v2 = mat[j][k];
9              fixLowers();
10             fixUppers();
11         }
12     }
13 }
```

Listing 3.8: Excerpt from implementation of MATRIX-UM-3

3.6 A lazy custom constraint

The implementation of constraints is a tradeoff between the number of nodes in the search tree (which is determined by consistency level) and the time spent at each node (which is dictated by the time to run the propagation algorithm). Hence even if a lazy constraint increases the size of a search tree it can still beat an eager constraint that takes a long time to run. This observation leads to the LAZY-UM-3 constraint that waits until two of the variables it acts over are instantiated before it does any propagation at all. This is what a forward checking constraint would do.

The constraint does nothing until it receives two notifications that variables have been instantiated. At this point the domains could be in one of three states enumerated

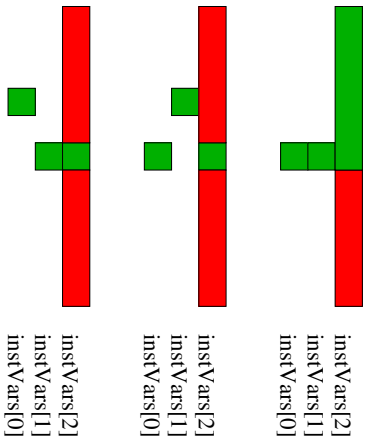


Figure 3.4: Cases in the analysis of Lazy-Um-3

in Figure 3.4, where in each case the two leftmost variables are instantiated; supported and unsupported values in the rightmost domain are coloured green and red respectively. Justification of this is by inspection bearing in mind the definition of Um. The constraint simply reduces the rightmost domain to contain only the shown supported values.

An $O(1)$ implementation of this is provided in Listing 3.9, but the following is a line-by-line account of the interesting implementation issues:

Lines 6-12 Ignore propagation events other than instantiations.

Lines 14-20 Function to return the number of instantiated variables. The reasoning behind the implementation of this function is surprisingly subtle. It might seem as though the number of instantiated variables could be tallied in a local variable, however the search algorithm could backtrack at anytime (thus un-instantiating a few variables) and the tally would become meaningless. Another tactic would be to use backtrackable storage which is restored to ancestral values when search backtracks, but this is fiddly and introduces overheads to the search process. Instead in this implementation the answer is worked out from scratch every time based on current variable state maintained by the search process.

Lines 43-62 Once two variables are instantiated, `awakeOnInst()` checks for the 3 cases shown in Figure 3.4 and does the propagation indicated in the diagram.

3.7 The TOOLKIT-UM-3 constraint revisited

The Toolkit-Um-3 constraint presented in Section 3.3, Custom-Um-3 from Section 3.4 and Matrix-Um-3 from Section 3.5 should all get the same answers, but how much benefit do the latter confer? In fact, we will show below that Toolkit-Um-3 does not enforce GBAC in any of the JChoco, ILog or ECLiPSe solvers. Hence a potentially large improvement in the number of nodes needed during search has been obtained. Furthermore in Chapter 5 empirical results will show that Custom-Um-3 also runs much faster at each node.

An example problem $x = \{1, 2, 3\}$, $y = \{2, 3\}$ and $z = \{3\}$ with constraint Toolkit-Um-3 was tried in the 3 toolkits listed above. No propagation happens in any of them. However when Custom-Um-3 is posted instead, $1 \in x$ is correctly removed by propagation.

```
1  public class LazyUm3 extends AbstractTernIntConstraint {
2      public LazyUm3(IntDomainVar v0, IntDomainVar v1, IntDomainVar v2) {
3          super(v0, v1, v2);
4      }
5
6      public void awake() throws ContradictionException { ; }
7      public void awakeOnBounds(int idx) throws ContradictionException { ; }
8      public void awakeOnInf(int idx) throws ContradictionException { ; }
9      public void awakeOnRem(int idx, int val) throws ContradictionException { ; }
10     public void awakeOnRemovals(int idx, IntIterator vals)
11                                             throws ContradictionException { ; }
12     public void awakeOnSup(int idx) throws ContradictionException { ; }
13
14     public int countInstantiated() {
15         int c = 0;
16         if(v0.isInstantiated()) c++;
17         if(v1.isInstantiated()) c++;
18         if(v2.isInstantiated()) c++;
19         return c;
20     }
21
22     //return the variables v0, v1 and v2 in an array with the instantiated ones
23     //before the uninstantiated ones
24     public IntDomainVar[] orderInst() {
25         IntDomainVar[] vs = new IntDomainVar[3];
26         int s = 0; //number of entries put at the beginning of array so far
27         int e = 0; //number of entried put at the end so far
28         if(v0.isInstantiated())
29             vs[s++] = v0;
30         else
31             vs[2-(e++)] = v0;
32         if(v1.isInstantiated())
33             vs[s++] = v1;
34         else
35             vs[2-(e++)] = v1;
36         if(v2.isInstantiated())
37             vs[s] = v2;
38         else
39             vs[2-e] = v2;
40         return vs;
41     }
42
43     public void awakeOnInst(int idx) throws ContradictionException {
44         //If zero or one variables are instantiated do nothing, if everything is
45         //instantiated then we must previously have made sure that the remaining
46         //domain was fine. However if 2 are instantiated then we ensure that
47         //everything in the last uninstantiated variable is permissible.
48         if(countInstantiated() == 2) {
49             IntDomainVar[] instVars = orderInst();
50             int i0 = instVars[0].getVal(); int i1 = instVars[1].getVal();
51             if(i0 > i1) {
52                 instVars[2].setInf(i1);
53                 instVars[2].setSup(i1);
54             } else if(i1 > i0) {
55                 instVars[2].setInf(i0);
56                 instVars[2].setSup(i0);
57             } else { //i0 == i1
58                 instVars[2].setInf(i0);
59             }
60         }
61     }
62 }
```

Listing 3.9: Implementation of Lazy-Um-3

In fact whatever level of consistency TOOLKIT-UM-3 enforces, it must be less than GBAC. This is becase each individual constraint in the model (like $<$ and `or`) is doing *no more* than BAC, and so they cannot do more than GBAC together. From above TOOLKIT-UM-3 has been observed doing doing less propagation than CUSTOM-UM-3, which does GBAC. Hence TOOLKIT-UM-3 must be doing strictly less than GBAC.

In fact, tests have shown that TOOLKIT-UM-3 *never* trims l.b.s under any circumstances, although it correctly trims u.b.s correctly in all cases. See Appendix A for the details.

Chapter 4

4 variable constraints

"Even a toad has four ounces of strength."— **Chinese proverb**

4.1 Introduction

As described in Section 2.5.2.2, in [Pro06] the species model from [GPSW03] was augmented so that the main ultrametric matrix M could hold ranges of values (corresponding to ranges of possible ancestral divergence dates) and the search variables changed to be a 3-D vector D of *decision variables*. In this model, for each $i < j < k$ there is a decision variable D_{ijk} which holds values describing the relationship between variables M_{ij}, M_{ik} and M_{jk}. The constraint posted to achieve this is as follows:

$$\begin{array}{rl} & d = 1 \Leftrightarrow v_1 > v_2 = v_3 \\ \wedge & d = 2 \Leftrightarrow v_2 > v_1 = v_3 \\ \wedge & d = 3 \Leftrightarrow v_3 > v_1 = v_2 \\ \wedge & d = 4 \Leftrightarrow v_1 = v_2 = v_3 \end{array} \tag{4.1}$$

Henceforth, this will be referred to as the UM-4 constraint. The intuition for the d values is to think of $\mathbf{1} \in d$ saying that v_1 is/can be greatest, $\mathbf{2} \in d$ saying that v_2 is/can be greatest, $\mathbf{3} \in d$ saying v_3 is/can be greatest, and finally $\mathbf{4} \in d$ representing the final possibility of them all being equal.

What this means is that a value in d is supported only when the v's have values that can achieve that relationship, e.g. 4 is supported if each of the 3 domains contain some value v. Conversely, a v variable is supported if it takes part in a relationship specified by one of the d values, e.g. v_1 can hold the distinct maximum out of v_1, v_2 and v_3 as long as d holds 1. The meaning of this in a species model is that the d variable records the species' relationship (e.g. m.r.c.a. 1 is more recent than m.r.c.a. 2 or 3) and the m variables hold ranges of dates when this m.r.c.a. could have diverged (e.g. m.r.c.a. 1 between times 1 and 3 and m.r.c.a 2 between 0 and 2). Since this is rather subtle some example are shown in Figure 4.1. In each case, the green shaded regions are supported, but the red regions are not. In 4.1(a), $4 \in d$ is unsupported because there is no value equal in all the domains, whereas in 4.1(b) this is not the case. In 4.1(c), the u.b. of v_2 is unsupported but in 4.1(d) is it supported by $2 \in d$.

If UM-4 is posted with $d = \{1, 2, 3, 4\}$ then it is equivalent to UM-3 as far as the v's are concerned.

Theorem 4 in Section 3.2 states that there are $\frac{1}{2}n(3n-1)$ distinct instantiations of the UM-3 constraint. This expression also gives the number of distinct instantiations of the UM-4 constraint because the v's uniquely determine d. The equation provides an informal method of testing the correctness of an implementation of the constraint.

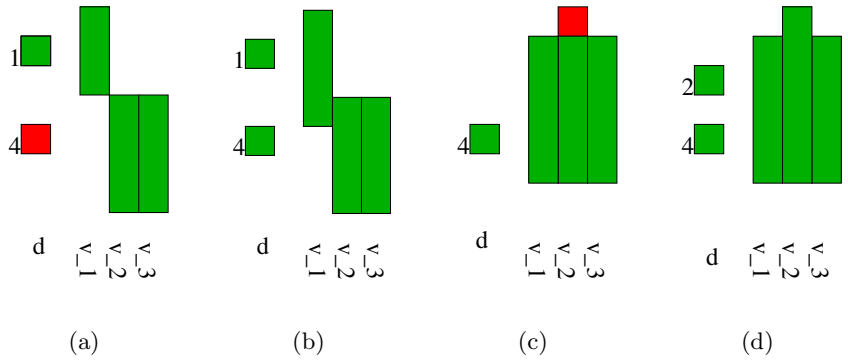


Figure 4.1: Some examples of supported and unsupported values in UM-4

```
1  public class ToolkitUm4 {
2      public static Constraint makeFourVar(IntDomainVar x, IntDomainVar y,
3                                            IntDomainVar z, IntDomainVar d) {
4          AbstractProblem pb = x.getProblem();
5          return pb.and(pb.ifOnlyIf(pb.eq(d, 1),
6                                    pb.and(pb.gt(x, y),
7                                           pb.and(pb.gt(x, z),
8                                                  pb.eq(y, z))),
9                        pb.ifOnlyIf(pb.eq(d, 2),
10                                   pb.and(pb.gt(y, x),
11                                          pb.gt(y, z),
12                                          pb.eq(x, z))),
13                       pb.ifOnlyIf(pb.eq(d, 3),
14                                   pb.and(pb.gt(z, x),
15                                          pb.gt(z, y),
16                                          pb.eq(x, y))),
17                       pb.ifOnlyIf(pb.eq(d, 4),
18                                   pb.and(pb.eq(x, y),
19                                          pb.eq(x, z),
20                                          pb.eq(y, z))));
21     }
22 }
```

Listing 4.1: Implementation of TOOLKIT-UM-4

4.2 The TOOLKIT-UM-4 implementation of UM-4

The baseline implementation based on Prosser's[1] is shown in Listing 4.1. It suffers from a similar problem to TOOLKIT-UM-3—extravagant memory usage (see Section 3.3). The same approach will be used in solving the problem this time around: a custom constraint.

Another motivation for the production of a custom constraint is that in the new species model it is essential for the UM-4 constraint to maintain a certain consistency level. With the old UM-3 version it didn't matter to the final result if the consistency level was less than GAC during search[2], because a final result was always fully instantiated. However now the v's may not be instantiated and so if they are inconsistent with d at the end then the result may be incorrect. Hence a UM-4 constraint suitable for use in the species model must guarantee a level of consistency such as GAC or GBAC which a toolkit constraint cannot, in general, guarantee.

[1] and improved immeasurably by the addition of LISP-style indentation!

[2] Although, as we said earlier, this issue is critical to execution speed.

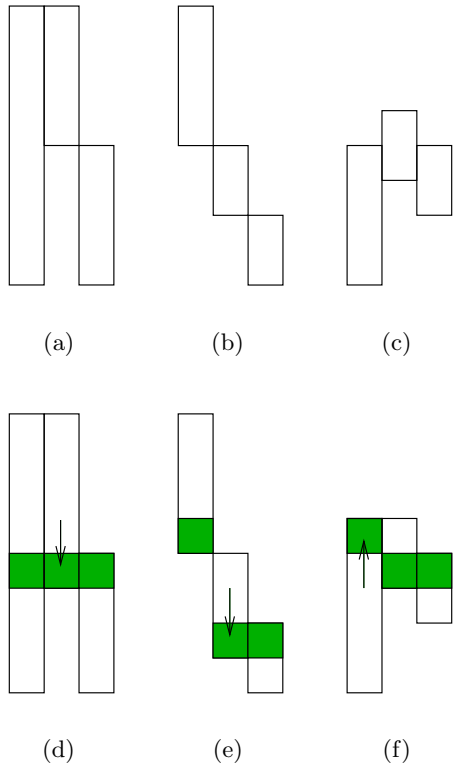


Figure 4.2: Scenarios where d=1 and d=4 are either unsupported or supported

4.3 The CUSTOM-UM-4 constraint

4.3.1 The interplay between d and the v's

Ensuring that the domain of d is supported w.r.t. the v's is relatively easy:

- 4 is unsupported if and only if any pair of domains is disjoint.
- 1 is unsupported if and only if v_2 and v_3 are disjoint (i.e. no equal pair) or else v_1 has nothing larger than a value in their intersection.
- 2 and 3 are unsupported symmetrically to 1.

To motivate this, Figure 4.2 shows pairs of scenarios, in each case the upper scenario shows when a d value would be unsupported but in the lower scenario a subtle difference in the v's provides it with support. In 4.2(a), the fact that v_2 and v_3 do not overlap rules out $4 \in d$, but in 4.2(d) each pair overlaps and so 4 is supported. In 4.2(b), $1 \in d$ is impossible because variables v_2 and v_3 are disjoint, but in 4.2(e) this has been rectified. In 4.2(c), variable v_1 is not bigger than anything in $v_2 \cap v_3$ so it can't be the largest of 3, but in 4.2(f) $1 \in d$ is supported.

The above reasoning translates directly into a procedure VToD (Listing 4.2 that can ensure everything in d is supported w.r.t. the current v's.

Making sure that the bounds of the v's are supported w.r.t. d is slightly more challenging. To motivate this assertion, observe that the whole of the previous chapter was concerned with propagating from d to v's in the special case when $d = \{1, 2, 3, 4\}$, i.e. when $x > y = z$, $y > x = z$, $z > x = y$ and $x = y = z$ are all allowed. Now the problem is to propagating when d is

```
1  if(any pair of v_1, v_2 or v_3 is disjoint)
2     remove 4 from d
3  else if(v_2 and v_3 are disjoint
4          or else v_1.UB <= min{v_2.UB, v_3.UB})
5     remove 1 from d
6  else if(v_1 and v_3 are disjoint
7          or else v_2.UB <= min{v_1.UB, v_3.UB})
8     remove 2 from d
9  else if(v_1 and v_2 are disjoint
10         or else v_3.UB <= min{v_1.UB, v_2.UB})
11    remove 3 from d
12 end if
```

Listing 4.2: Pseudocode for procedure VToD checking d support

$$
\begin{array}{cccccccccccc}
 & \{1,2,3,4\} & & & & & & & & & & \\
\text{or} & \{1,2,3\} & \text{or} & \{1,2,4\} & \text{or} & \{1,3,4\} & \text{or} & \{2,3,4\} & & & & \\
\text{or} & \{1,2\} & \text{or} & \{1,3\} & \text{or} & \{1,4\} & \text{or} & \{2,3\} & \text{or} & \{2,4\} & \text{or} & \{3,4\} \\
\text{or} & \{1\} & \text{or} & \{2\} & \text{or} & \{3\} & \text{or} & \{4\} & & & &
\end{array}
$$

These are $\mathrm{C}(4,4)+\mathrm{C}(4,3)+\mathrm{C}(4,2)+\mathrm{C}(4,1) = 1+4+6+4 = 15$ different possibilities[3]. Taking experience from Chapter 3 as a guide, where 10s of cases were used to analyse just one of these, it would seem that analysing all of these cases would be quite time consuming and potentially error prone. Instead, the approach taken is to consider 1,2,3 and 4 in turn and to imagine for each that it is *the sole* value in d. Now we can work out the greatest upper bounds and the least lower bounds that the value can support. If d happens to be $\{1,3,4\}$ then the upper bounds actually supported is the maximum out of those found earlier for 1, 3 and 4, and the lower bound supported is the minimum out of those found earlier. For upper bounds, this is because any greater bound is not supported by d, and any lesser bound removes a value supported by d. The argument for lower bounds is symmetric.

An example of this whole process is shown in Figure 4.3. From left to right the top part shows:

- that $d = \{1,3,4\}$ and the initial bounds of the vs (unshaded);
- the bounds supported by each of 1,2,3 and 4; and finally,
- the actual supported bounds by taking the greatest upper bound and the least lower bound.

The bottom part shows in gold the ranges supported in the v variables by 1, 3 and 4 individually; and in green the bound supported by all three together. The red ranges are the bounds that are supported by $d = \{2\}$, but these are ignored since $2 \notin d$ in this example.

This reasoning translates directly into a procedure DToV (Listing 4.3) that can propagate a change in d to the v's. It runs in $O(1)$ time.

[3]In fact they comprise the powerset $P(\{1,2,3,4\})$ minus ϕ. See [Ros98] for a definition.

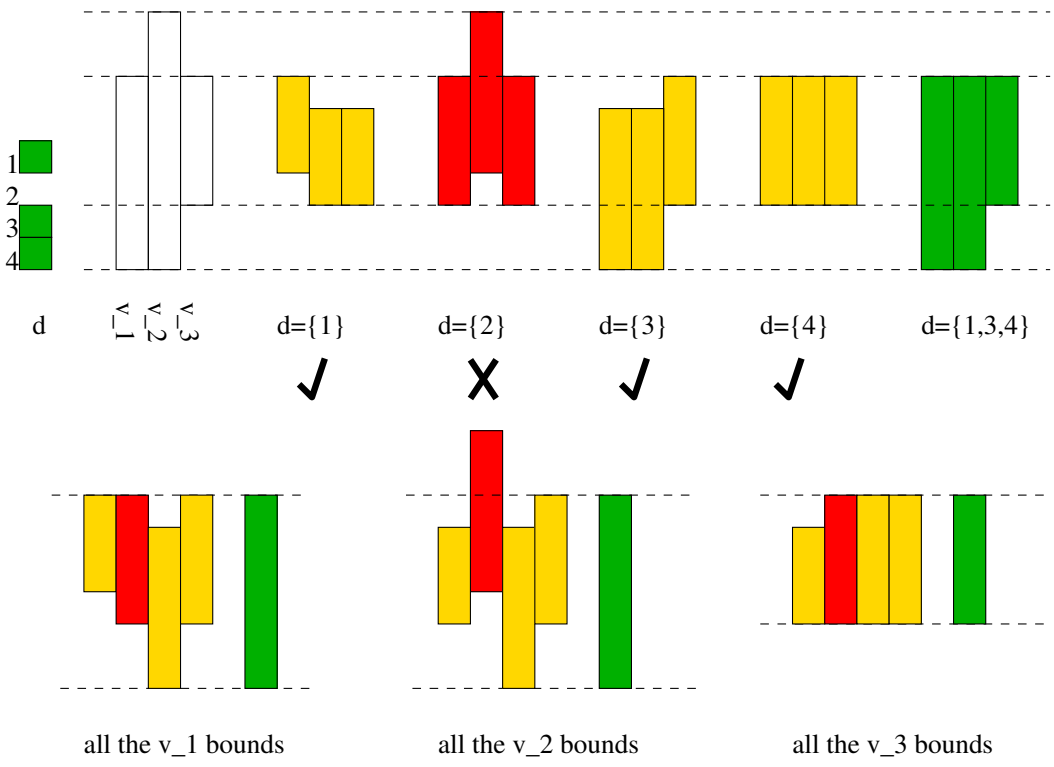


Figure 4.3: An example of checking support for M variables by case analysis

```
1  newV1UB = negative infinity //any u.b. is greater than this
2  newV2UB = negative infinity
3  newV3UB = negative infinity
4  newV1LB = infinity          //any l.b. is less than this
5  newV2LB = infinity
6  newV3LB = infinity
7  if(d contains 4)
8     expr1 = min{v1.UB, v2.UB, v3.UB} //least upper bound
9     expr2 = max{v1.LB, v2.LB, v3.LB} //greatest lower bound
10    newV1UB = max{newV1UB , expr1}
11    newV2UB = max{newV2UB , expr1}
12    newV3UB = max{newV3Ub , expr1}
13    newV1LB = min{newV1LB , expr2}
14    newV2LB = min{newV2LB , expr2}
15    newV3LB = min{newV3LB , expr2}
16 else if(d contains 1)
17    expr1 = max{v2.LB, v3.LB}
18    newV1LB = min{newV1LB , expr1 + 1}
19    newV2LB = min{newV2LB , expr1}
20    newV3LB = min{newV3LB , expr1}
21    expr2 = min{v2.UB, v3.UB, v1.UB - 1}
22    newV1UB = max{newV1UB , v1.UB}
23    newV2UB = max{newV2UB , expr2}
24    newV3UB = max{newV3UB , expr2}
25 else if(d contains 2)
26    //symmetric with (d contains 1) branch
27 else if(d contains 3)
28    //symmetric with (d contains 1) branch)
29 end if
30 v1.LB = newV1LB
31 v2.LB = newV2LB
32 v3.LB = newV3LB
33 v1.UB = newV1UB
34 v2.UB = newV2UB
35 v3.UB = newV3UB
```

Listing 4.3: Pseudocode for procedure DToV checking v's support

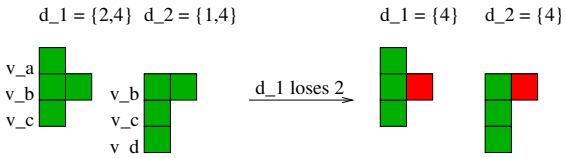


Figure 4.4: A reduction in search space as a result of AC

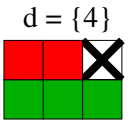
4.3.2 Overall constraint design

A first idea that would give correct answers in the species model would be to propagate exclusively from d to v_1, v_2 and v_3. The problem with this is that the whole space of D variables would have to be searched because no values are ruled out by propagation—a level of consistency very close to forward checking and consequently with a danger of thrashing.

A better constraint propagates in both directions: from d to v_1, v_2 and v_3; and vice-versa. In the species tree model, where constraints overlap, there will be a benefit to the search process whenever a choice for d leads indirectly to the reduction in the domain of another d (e.g. the example of Figure 4.4, where the loss of 2 from d_1 causes a reduction in the domain of a different d variable).

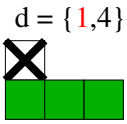
Before presenting a constraint that does the complete job we will analyse a few scenarios requiring propagation in order to reason about when it will be needed:

Scenario 1 - A change in v_1, v_2 or v_3 causes loss of support in v_1, v_2 or v_3 This can certainly happen:

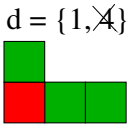


In this diagram and those below the value marked X has been lost, the green shaded values retain support but the red shaded values have lost support.

Scenario 2 - A change in v_1, v_2 or v_3 causes loss of support in d Example:



Scenario 3 - A change in d causes loss of support in v_1, v_2 or v_3 Example:



Scenario 4 - A change in d causes loss of support in d This is impossible. The semantics of the different values of d are such that they are disjoint and the removal of one possibility cannot lead directly to another being ruled out.

These 4 scenarios show that a change in d can only result in the need to check values in v_1, v_2 and v_3 for support; but a change in v_1, v_2 and v_3 can result in the need to check both d and variables themselves. Hence the following CUSTOM-UM-4 propagation algorithm (Listing 4.4).

```
1  do DToV
2  if(propagation event is on domain of v_1, v_2 or v_3)
3     do VToD
```

Listing 4.4: Pseudocode for propagation algorithm Custom-Um-4

The level of consistency that this constraint maintains is a hybrid between GAC and GBAC: in d the individual values must be supported but only the bounds of v_1, v_2 and v_3 need to be supported.

The argument in this Section amounts to a proof of correctness and GBAC status for the Custom-Um-4 constraint, although it is rigorous it is presented in a less formal way than the proof of the previous Chapter.

4.3.3 Constraint implementation

The implementation of Custom-Um-3 is a trivial translation of the above pseudocode into Java. For completeness, however, it is listed in Listing C.2 of Appendix C.

4.3.4 Generalising this technique

The Custom-Um-4 constraint could be abstracted as having a d variable that says which clauses in a disjunction are enabled. The technique in this Section solves the problem by considering each clause separately and propagating the whole constraint by taking the most inclusive bounds for those that are enabled. The d values are ruled out by a reverse process.

Other constraints doing a similar job could be implemented the same way. For example the disjunction/`or` constraint itself works this way. When posting `or` introduce an auxiliary non-search $\{0, 1\}$ variable d. When the 1st clause is ruled out ensure $0 \notin d$. When the 2nd clause is ruled out ensure $1 \notin d$. To trim the variables in the disjunction, take the union of the supported values for the clauses corresponding to values still in d. If this procedure was used in the JChoco implementation of `or` then it would improve propagation. For example, the problem $x = \{1, 2\}$ with constraint `x=1 or x=1` currently does no propagation, but a new implementation could remove 2 from the domain of x. If a constraint like this had been used in Toolkit-Um-3 then it might have done full GBAC propagation, although it would still have been slow and bulky.

4.4 Matrix version

The new species model posts the constraint Um-4 in almost the same way as the old one posted Um-3. Hence it will benefit in the same way as before from a matrix formulation (see Section 3.5). Such a constraint has been implemented and is listed in Listing C.3 in Appendix C. Since its design is similar to that of Section 3.5 and because it will not be used in the later empirical study it is not described in any detail here. There was a level of challenge in the implementation but it was very much analagous to the earlier matrix constraint. This constraint also does propagation in $O(n)$ time.

Chapter 5

Empirical study

"Mathematics alone makes us feel the limits of our intelligence. For we can always suppose in the case of an experiment that it is inexplicable because we don't happen to have all the data. In mathematics we have all the data ... and yet we don't understand."— **Simone Weil**

5.1 Birds empirical study

The previous chapters have presented constraint designs and proved properties about them such as correctness and consistency level. However, the aim of this project was to provide an improvement in time and space in the species model, so we will now turn our attention to establishing if the implementations of the constraints make any practical difference. This empirical study should also bolster confidence in properties like correctness and consistency levels that we have previously proved, but not tried!

5 different Um constraints as well as an imperative solution were used in the study. Table 5.1 lists short names that will be used throughout this Chapter along with references to where the constraint was introduced.

Predictions:

1. The number of nodes needed for Custom-Um-3, Matrix-Um-3 and Custom-Um-4[1] should be same in all cases, since they maintain the same GBAC consistency level.

2. If any constraint finds a solution they all should, since they are all believed to be correct.

[1]The full capabilities of this constraint are not being exploited, since only the v's are being searched, but it's better to do a little testing than none at all!

Short name	Reference
Imperative	Section 2.5
Toolkit-Um-3	Section 3.3
Custom-Um-3	Section 3.4
Matrix-Um-3	Section 3.5
Lazy-Um-3	Section 3.6
Custom-Um-4	Section 4.3

Table 5.1: Constraints used in study

3. The number of nodes needed for Lazy-Um-3 and Toolkit-Um-3 should be at least as large as for the others, since they maintain a lower consistency level.

4. The memory usage for the constraints should be (in ascending order): Matrix-Um-3; tie between Lazy-Um-3 and Custom-Um-3; Custom-Um-4; and finally Toolkit-Um-3. This is due to the fact that Matrix-Um-3 uses one Java object in total; Lazy-Um-3 and Custom-Um-3 use $C(n,3)$ Java objects; Custom-Um-4 use $C(n,3)$ but there are $C(n,3)$ additional d variables in the model; and because Toolkit-Um-3 is posted $C(n,3)$ times but uses 23 times the space.

5. If Lazy-Um-3 can ever match the others' nodes, it should finish faster because it does very little work at each one.

6. Build time should be roughly proportional to the memory size of a problem.

5.1.1 Results

The experiments were run under the conditions described in Section E.1. The min-domain variable ordering heuristic was used.

The following table lists the results of the experiment on seabird data taken from [KP02]. The columns are as follows: "Data" lists the identifiers for the input trees, n is the number of species in total for all the input trees, "Constraint" is the constraint used, "Build" is the time taken to load the model, "Solve" is the time taken to find a first solution or to prove none exist, "Total" is the sum of "Build" and "Solve", "Nodes" is the number of nodes in the search tree (see [BZF04]) and "Mem" is the model memory size in MB. In Section E.2 there are some notes on the sources of these data.

Data	n	Constraint	Sol^n	Build	Solve	Total	Nodes	Mem
AB	23	Toolkit-Um-3	T	2056	374	2447	23	26.92
		Lazy-Um-3	T	294	19667	19978	7706	0.98
		Custom-Um-3	T	293	139	449	23	0.98
		Matrix-Um-3	T	183	131	331	23	0.24
		Custom-Um-4	T	455	420	893	23	3.84
		Imperative	T			13		
AC	32	Toolkit-Um-3	F	2670	327	3014	0	36.34
		Lazy-Um-3	F	320	77	414	0	1.26
		Custom-Um-3	F	320	160	497	0	1.26
		Matrix-Um-3	F	189	153	359	0	0.34
		Custom-Um-4	F	521	350	889	0	5.14
		Imperative	F			12		
AD	47	Toolkit-Um-3	T	8235	946	9199	38	118.51
		Lazy-Um-3	T	550	165	733	38	3.81
		Custom-Um-3	T	549	261	827	38	3.81
		Matrix-Um-3	T	220	248	486	38	0.70
		Custom-Um-4	T	1128	982	2128	38	16.45
		Imperative	T			22		
AE	95	Toolkit-Um-3	DNL	DNL	DNL	DNL	DNL	> 629
		Lazy-Um-3	F	3174	316	3508	0	30.64
		Custom-Um-3	F	3171	1558	4748	0	30.64
		Matrix-Um-3	F	340	1477	1836	0	2.79
		Custom-Um-4	F	8306	5427	13752	0	13.88
		Imperative	F			37		

AF	31	Toolkit-Um-3	T	2497	379	2893	19	32.99
		Lazy-Um-3	T	463	129	609	24	1.16
		Custom-Um-3	T	313	158	489	18	1.16
		Matrix-Um-3	T	188	137	342	18	0.32
		Custom-Um-4	T	501	514	1032	18	4.67
		Imperative	T			20		
AG	46	Toolkit-Um-3	T	7671	871	8560	31	111.07
		Lazy-Um-3	T	532	161	711	31	3.61
		Custom-Um-3	T	531	268	817	31	3.61
		Matrix-Um-3	T	222	252	491	31	0.68
		Custom-Um-4	T	1077	993	2088	31	15.45
		Imperative	T		21			
BC	29	Toolkit-Um-3	F	2056	21931	24004	171	26.90
		Lazy-Um-3	F	285	285207	285508	265237	0.97
		Custom-Um-3	F	286	116	419	0	0.97
		Matrix-Um-3	F	171	107	295	0	0.27
		Custom-Um-4	F	448	277	743	0	3.82
		Imperative	F			8		
BD	42	Toolkit-Um-3	T	5833	930	6780	33	84.26
		Lazy-Um-3	T	441	DNF	DNF	DNF	2.75
		Custom-Um-3	T	438	265	720	33	2.75
		Matrix-Um-3	T	201	251	469	33	0.55
		Custom-Um-4	T	875	1003	1896	33	11.72
		Imperative	T			17		
BE	94	Toolkit-Um-3	DNL	DNL	DNL	DNL	DNL	> 629
		Lazy-Um-3	DNF	3117	DNF	DNF	DNF	29.82
		Custom-Um-3	F	3363	23377	26758	0	29.82
		Matrix-Um-3	F	335	16340	16693	0	2.71
		Custom-Um-4	F	7990	91936	99945	0	134.61
		Imperative	F			11		
BF	30	Toolkit-Um-3	T	2405	343	2765	29	29.83
		Lazy-Um-3	T	295	83	395	29	1.05
		Custom-Um-3	T	293	114	424	29	1.05
		Matrix-Um-3	T	174	99	290	29	0.28
		Custom-Um-4	T	466	397	880	29	4.22
		Imperative	T			8		
BG	40	Toolkit-Um-3	T	5098	651	5766	30	72.71
		Lazy-Um-3	T	413	136	567	31	2.32
		Custom-Um-3	T	413	214	645	30	2.32
		Matrix-Um-3	T	203	353	574	30	0.51
		Custom-Um-4	T	776	829	1622	30	10.03
		Imperative	T			13		
CD	45	Toolkit-Um-3	T	10056	1134	11208	45	143.91
		Lazy-Um-3	T	608	186	812	45	4.51
		Custom-Um-3	T	609	294	921	45	4.51
		Matrix-Um-3	T	224	276	518	45	0.77
		Custom-Um-4	T	1324	1154	2496	45	19.82
		Imperative	T			14		
CE	68	Toolkit-Um-3	DNL	DNL	DNL	DNL	DNL	> 629
		Lazy-Um-3	T	3071	564	3653	72	29.82

		Custom-Um-3	T	3080	1545	4643	68	29.82
		Matrix-Um-3	T	516	1451	1985	68	2.72
		Custom-Um-4	T	7929	7760	15707	68	134.61
		Imperative	T			36		
CF	34	Toolkit-Um-3	T	3101	563	3681	30	43.72
		Lazy-Um-3	T	328	124	469	32	1.47
		Custom-Um-3	T	424	152	593	30	1.47
		Matrix-Um-3	T	180	133	331	30	0.36
		Custom-Um-4	T	561	522	1101	30	6.14
		Imperative	T			11		
CG	44	Toolkit-Um-3	F	6683	587	7288	0	97.10
		Lazy-Um-3	F	641	84	743	0	3.22
		Custom-Um-3	F	494	229	740	0	3.22
		Matrix-Um-3	F	210	215	442	0	0.61
		Custom-Um-4	F	980	697	1695	0	13.56
		Imperative	F			14		
DE	104	Toolkit-Um-3	DNL	DNL	DNL	DNL	DNL	> 629
		Lazy-Um-3	F	4041	249	4308	0	38.66
		Custom-Um-3	F	4051	1960	6029	0	38.66
		Matrix-Um-3	F	360	2021	2400	0	3.31
		Custom-Um-4	F	10664	7454	18136	0	180.87
		Imperative	F			34		
DF	44	Toolkit-Um-3	T	6613	987	7617	37	97.10
		Lazy-Um-3	DNF	484	DNF	DNF	DNF	3.21
		Custom-Um-3	T	628	270	915	37	3.21
		Matrix-Um-3	T	203	250	470	37	0.60
		Custom-Um-4	T	970	940	1928	37	13.55
		Imperative	T			17		
DG	56	Toolkit-Um-3	F	14090	2280	16388	1	201.42
		Lazy-Um-3	DNF	800	DNF	DNF	DNF	6.30
		Custom-Um-3	F	791	674	1482	0	6.30
		Matrix-Um-3	F	252	640	910	0	0.99
		Custom-Um-4	F	1825	2491	4334	0	27.96
		Imperative	F			19		
EF	94	Toolkit-Um-3	DNL	DNL	DNL	DNL	DNL	> 629
		Lazy-Um-3	DNF	3070	DNF	DNF	DNF	29.82
		Custom-Um-3	F	3070	13299	16387	0	29.82
		Matrix-Um-3	F	331	9546	9896	0	2.71
		Custom-Um-4	F	7906	51835	59760	0	134.61
		Imperative	F			12		
EG	97	Toolkit-Um-3	DNL	DNL	DNL	DNL	DNL	> 629
		Lazy-Um-3	F	3424	249	3690	0	32.31
		Custom-Um-3	F	3340	11647	15005	0	32.31
		Matrix-Um-3	F	344	8900	9262	0	2.89
		Custom-Um-4	F	8763	45744	54526	0	147.49
		Imperative	F			15		
FG	38	Toolkit-Um-3	DNF	4299	DNL	DNL	DNL	61.41
		Lazy-Um-3	DNF	381	DNL	DNL	DNL	2.04
		Custom-Um-3	F	387	233	637	0	2.04
		Matrix-Um-3	F	195	212	424	0	0.46
		Custom-Um-4	F	705	665	1388	0	8.62

		Imperative	F		10			
ABDF	72	Toolkit-Um-3	T	27032	5291	32340	63	382.52
		Lazy-Um-3	DNF	1320	DNF	DNF	DNF	11.82
		Custom-Um-3	T	1303	762	2083	59	11.82
		Matrix-Um-3	T	277	722	1017	59	1.48
		Custom-Um-4	T	3238	3462	6718	59	52.69
		Imperative	T			34		
ABDG	78	Toolkit-Um-3	DNF	60847	DNF	DNF	DNF	553.49
		Lazy-Um-3	DNF	1754	DNF	DNF	DNF	16.44
		Custom-Um-3	F	1764	4476	6258	0	16.44
		Matrix-Um-3	F	301	3633	3952	0	1.91
		Custom-Um-4	F	4806	18482	23303	0	75.81
		Imperative	F			29		
ACDF	72	Toolkit-Um-3	F	31067	1931	33015	0	434.84
		Lazy-Um-3	F	1442	164	1624	0	13.41
		Custom-Um-3	F	1446	985	2449	0	13.41
		Matrix-Um-3	F	286	649	953	0	1.61
		Custom-Um-4	F	3616	2103	5734	0	60.02
		Imperative	F			28		
ACDG	81	Toolkit-Um-3	DNL	DNL	DNL	DNL	DNL	> 629
		Lazy-Um-3	F	2049	270	2336	0	18.13
		Custom-Um-3	F	2018	2024	4060	0	18.13
		Matrix-Um-3	F	307	1711	2036	0	2.06
		Custom-Um-4	F	5142	7523	12683	0	84.86
		Imperative				35		
ACE	97	Toolkit-Um-3	DNL	DNL	DNL	DNL	DNL	> 629
		Lazy-Um-3	F	3366	351	3735	0	32.32
		Custom-Um-3	F	3342	1710	5070	0	32.32
		Matrix-Um-3	F	737	1632	2387	0	2.91
		Custom-Um-4	F	8965	5970	14953	0	147.51
		Imperative	F			38		

The inputs and output from instance ABDF have been printed in Appendix B. All the constraints as well as the imperative solution get the same result up to symmetry.

5.1.2 Interpretation of results

These results could be summarised as follows:

- Matrix-Um-3 is extremely successful in reducing memory usage compared to Toolkit-Um-3, achieving up to a factor of 290 (for ABDG) improvement in memory usage. This improvement also leads to a large reduction in the overall time to load problems.
- Custom-Um-3 and Matrix-Um-3 reduce search time compared to Toolkit-Um-3, always beating it by a factor of 3 or better. Matrix-Um-3 is a little faster than Custom-Um-3.
- Matrix-Um-3 improves total time by a large factor over Toolkit-Um-3, due to the added effects of reduced load and solve times.

- In problems AB, AF, BC, BG, CE, CF, DG and ABDF, MATRIX-UM-3, CUSTOM-UM-3 or CUSTOM-UM-4 were observed to solve instances using fewer nodes than either TOOLKIT-UM-3 or LAZY-UM-3. This is consistent with the former maintaining a higher level of consistency than the latter.
- CUSTOM-UM-3, MATRIX-UM-3 and CUSTOM-UM-4 always use the same number of nodes. This is consistent with them maintaining the same GBAC consistency level.
- All solutions get the same answer, increasing confidence in their correctness.
- LAZY-UM-3 runs very quickly when it uses the same number of nodes as the others, but occasionally it takes a very large number of nodes and a long time.

5.1.3 Conclusions

It would be hard to make a case for continuing to use TOOLKIT-UM-3 rather than CUSTOM-UM-3 or MATRIX-UM-3, since these occupy much less space and are faster. MATRIX-UM-3 in particular represents a huge improvement in both space and time. The memory usage grows so slowly that any foreseeable species data could now be loaded into the memory of a typical desktop computer.

The CUSTOM-UM-4 constraint is not specialised for use in the above model, so we should not write it off due to its poor performance, but rather first try it in an experiment that exploits its full generality. However this experiment provides evidence that it does indeed maintain GBAC.

It is slightly less easy to throw out LAZY-UM-3 on account of the fact that it is very often faster than CUSTOM-UM-3. However when LAZY-UM-3 does badly it does extremely badly and when it does well it wins by a small constant factor. This is because it cannot propagate any better than GBAC and hence it cannot win by more than a constant factor, however it could potentially lose by an exponential factor; this would make it extremely risky to use LAZY-UM-3 until more is known about the circumstances when it does badly.

The CP solution does not yet appear to be competitive with the imperative solution in a simple-minded speed comparison, although the gap has been closed dramatically. All the flexibility and easy of modelling of the CP model is retained in the new version, so it still well worth working on. A similar study was carried out in [Pro06] when 6 of the 21 pairs of birds files could not be loaded, now all pairs can be loaded successfully.

5.1.4 Re-running the experiment

BASH script `birdsrun.sh` in the `empirical` directory of the attached CD will repeat the experiment and store the results at path `birds_data/runs`. The output format of these run files is:

```
1 [memory in bytes] [build time in ms] [run time in ms] [total time] [solvable?]
2 [nodes] [backtracks]
3
4 [actual solution tree]
```

Listing 5.1: Output format of birds experiment run files

5.2 Random empirical study

5.2.1 Motivation

The tests on seabird data from the previous Section have shown how the constraints produced have improved on previous work in terms of reduced nodes, time and memory usage. These results on specific instances raise a few interesting general questions:

- How does the model scale, on average, as a function of the size of the instance (in this case, defined to be the number of leaf nodes) in terms of time, nodes, memory usage, etc.?
- What is the relative performance of the different types of constraint, on the average. Can any general relationships be found, e.g. does the matrix constraint always solve a problem in fewer nodes than the primitive constraint?
- Do the constraints differ in their ability to make short work of easy or hard instances?
- How do variable ordering heuristics affect results?

These questions will be addressed in this section.

As was stated in previous chapters, the constraints maintain different levels of consistency and they all propagate in constant time, however this tells us relatively little in the general case about their behaviour on problems. For example, it may be that a lower level of consistency done quickly will result in a faster solution overall, if the decrease in effort at each node dominates the increase in the number of nodes.

These issues showed up in instance BG in the previous Section when it was solved more quickly by Lazy-Um-3 using *more* nodes. Nevertheless since specific instances may exhibit pathological or trivial nature; it is hard to tell if they are representative without investigating the average case. It might be possible to take the approach of working out analytically an expression for the asymptotic time, space or nodes averaged over all instances of a problem. However here the approach taken is to investigate the average case by averaging behaviour over a set of random instances. This does not constitute a proof of behaviour, but it can strongly suggest that certain relationships are present.

The problem that will be generated and solved is that of constructing the unique (subject to provisos described later) ultrametric matrix that satisfies the triples produced from a *single* random tree by the BreakUp algorithm. Random trees with different numbers of leaves will be generated, and hence different numbers of variables and triples are needed in the model. This problem is simple but non-trivial and all of the constraints in this report can be applied to it. It is closely related to the species tree problem—the difference is that in the species tree problem the triples are sourced from two or more different trees where there is no guarantee of either a unique solution or any solution at all.

Due to a symmetry in the model used, if the random input tree does not have maximum depth for its number of leaves then multiple ultrametric matrices could be a correct result but these multiple solutions must be trivially related to one another. To justify this, all the solutions to two example problems with 4 leaves are shown in Figure 5.1. 5.1(a) is a tree with the unique solution 5.1(c). 5.1(b) is a tree with multiple solutions 5.1(d). The reason why 5.1(b) has multiple solutions is that it has less than the maximum permitted depth of 3, so that multiple labels can be assigned to its branch nodes. Conversely 5.1(a) has a single solution because it has the maximum permitted depth.

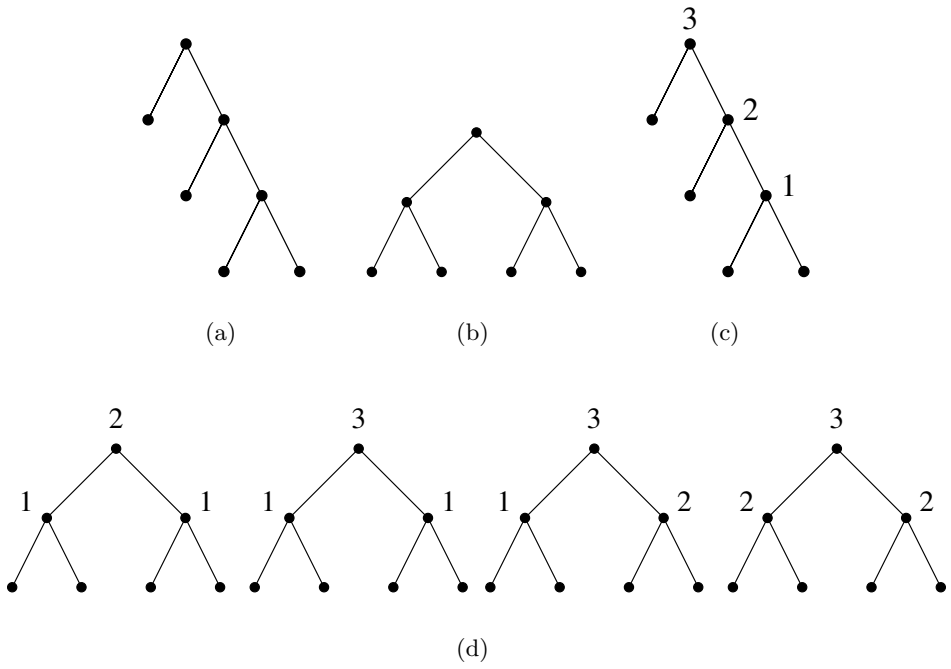


Figure 5.1: All UM trees with 4 leaves

In the results this issue has been ignored, with only time and nodes to the *first* solution being obtained. However instances with different numbers of solutions are intuitively of a likeness, because they seem (given without proof) minimally solvable, in the sense that the addition of any more non-trivial triples would make them unsolvable and the removal of any triples would result in multiple and distinct solutions. It is future work to gain a better understanding of how solvability changes according to the set of triples and to investigate phase transition behaviour in the problem[CKT91].

All the following experiments were initially carried out with the min-domain dynamic variable ordering heuristic. This was misleading because the variables could have been instantiated in a different order for different constraints, meaning that rival constraints may not be afforded the opportunity to perform equally well if the heuristic makes bad decisions on their behalf. As a result, the experiments were repeated with an arbitrary static variable ordering, but both results are presented.

5.2.2 Procedure

1. Generate at random 50 bifurcating trees for each of 4,8,...,44 leaves.
2. Run model using the triples extracted from each random tree by the BREAKUP algorithm for each different constraint type. Record time to solve, time to build model, nodes to solve and memory used for model.
3. Analyse data.

5.2.2.1 Notes on step 1

A random bifurcating tree with k leaves can be generated recursively. The only tree with 1 leaf is the tree with 1 node. A tree with $k > 1$ leaves can have subtrees of size 1 and $k-1$, 2 and $k-2$ or so on to $\lfloor k \rfloor$ and $\lceil k \rceil$. Hence to generate a random tree

```
1  public class RandomBTree {
2      private static int nextLabel = 0;
3
4      //print a random tree with n leaf nodes, in newick format, use nextLabel to
5      //choose a unique label for the leaf
6      public static void printRandomBTree(int n) {
7          if(n == 1) {
8              System.out.print("species" + nextLabel);
9              nextLabel++;
10         } else {
11             //take the smaller random number of leaves to go in the left
12             //subtree, the balance go in the right subtree
13             int split = 1 + (int)((k / 2) * Math.random());
14             System.out.print("(");
15             printRandomBTree(split);
16             System.out.print(",");
17             printRandomBTree(n - split);
18             System.out.print(")");
19         }
20     }
21
22     public static void main(String[] args) {
23         printRandomBTree(Integer.parseInt(args[0]));
24         System.out.println(";");
25     }
26 }
```

Listing 5.2: Code to generate random bifurcating tree

pick a random integer f from 1 through to $\lfloor k \rfloor$ and from this calculate another integer $s = k - f$. These are the sizes of the two subtrees and they are generated recursively.

Java code to generate trees in Newick format[Ols90] is shown in Listing 5.2. Newick format intrinsically orders subtrees, but this code does not take advantage of the facility because it always puts the smaller subtree first.

5.2.2.2 Notes on step 2

The statistics collected were obtained from the sources described in Section E.2.

It was observed that solution time is quite variable for the same instance run under apparently identical conditions. This could be due to background and system tasks being scheduled to run, garbage collection, scheduling variation, caching variation, etc. To remove outliers each instance was executed 5 times and the result used is the median of the 5 runs. This variation only applies to time but not nodes or memory.

5.2.2.3 Notes on step 3

Shell scripts were used to process the data and to extract results.

5.2.2.4 Access to data and scripts

The data and scripts are available on the attached CD. The procedure to run the scripts is in Appendix E.3. Following these instructions would repeat the experiment, but they take several days to complete because of the repetition involved in filtering outliers.

5.2.2.5 Memory usage

One of the aims of this project was to reduce memory usage to allow larger instances to be modelled. Figure 5.2 shows how the model memory usage grows with increasing number of leaves. The new constraints are a significant improvement on the Toolkit-Um-3 benchmark.

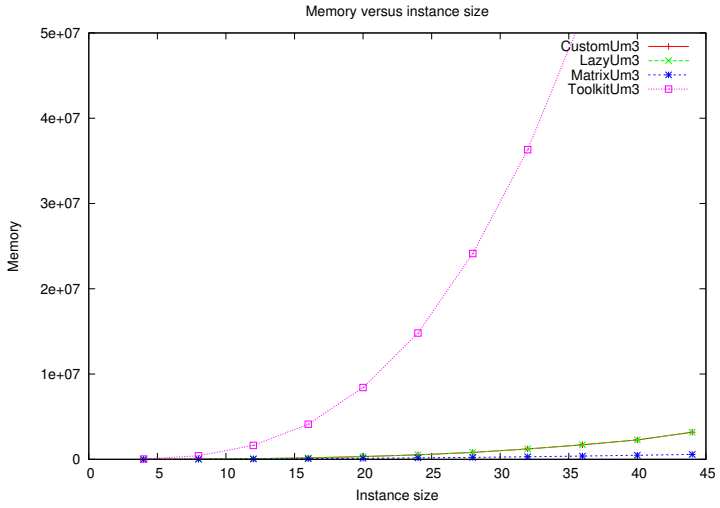


Figure 5.2: Random study: memory versus size

In the creation of the JCHOCO model of this type of problem:

1. $\frac{n(n-1)}{2}$ `IntDomainVar` objects are created to represent variables and domains;
2. either 1 (for MATRIX-UM-3) or $\frac{n(n-1)(n-2)}{6}$ (all other constraints) `Constraint` objects are posted to constrain the matrix to be UM;
3. $n-2$ `Constraints` are posted to describe the triples derived from the input tree by BREAKUP, each of which consists of a constant number of nested `Constraints`;
4. a constant number of other objects like those implementing variable ordering heuristics; and
5. any other data that JCHOCO decides to create in the process of variable and constraint creation.

The `IntDomainVar` objects are bounds variables and so their representation should have constant size, hence the memory occupied by variables should be $O(n^2)$. The `Constraint` objects each maintain a constant amount of state and so the memory occupied should be $O(1)$ for MATRIX-UM-3 and $O(n^3)$ for the others. The variable ordering heuristic should occupy $O(n^2)$ space because it requires an array of the $O(n^2)$ instantiated variables. The amount of space taken up by JCHOCO's internal processes has not been investigated, but it should only inflate figures by a constant factor. Hence overall the model should have a space complexity of $O(n^2)$ for the matrix constraint and $O(n^3)$ for the others. Consistent with expectations, the memory sizes using CUSTOM-UM-3 and LAZY-UM-3 are identical.

The plot of the log of memory versus problem size (Figure 5.3) appears to exhibit a logarithmic behaviour (as a consequence of $\log(x^b) = b\log(x)$), which is consistent with the belief that the model memory is increasing polynomially with the problem size. The hypothesis that the model is $O(n^2)$ or $O(n^3)$ is very strong, but it's hard to be certain without investigating every detail of JCHOCO's internal representation.

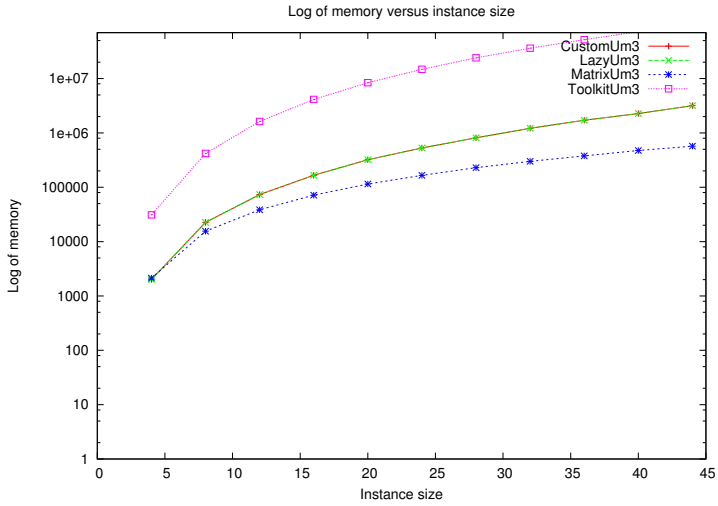


Figure 5.3: Random study: log of memory versus size

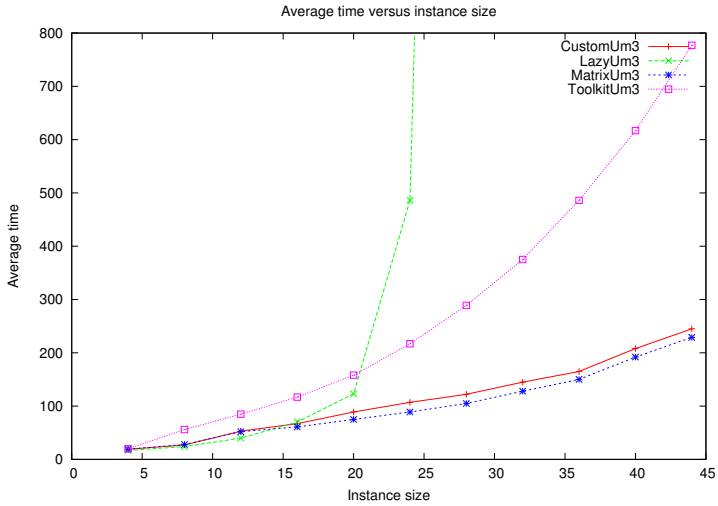


Figure 5.4: Random study: average time versus size for static variable order

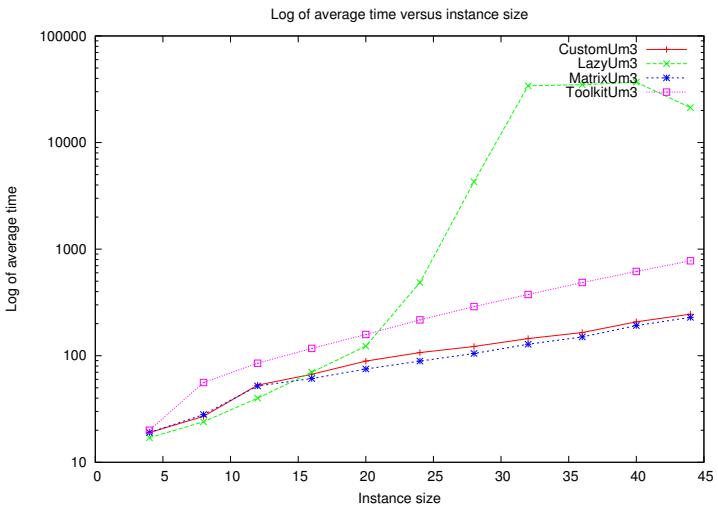


Figure 5.5: Random study: log of average time versus size for static variable order

5.2.2.6 Experiments with static ordering

Consider the timing results[2] in Figure 5.4. As expected from the experiments on species trees, Matrix-Um-3 does the best on average, but Custom-Um-3 is competitive. Toolkit-Um-3 is much slower than them. Lazy-Um-3 performs even slower than Toolkit-Um-3.

Figure 5.5 gives more feeling for the rate of growth in time. Since the y-axis is logarithmic, and the curves look like they might be logarithmic (they look like the memory growth curves in Figure 5.2 which definitely *should* be logarithmic, for the reason described in Section 5.2.2.5), it seems fair to conclude that the rate of growth in time is polynomial. This is because it cannot conceivably be sub-linear for information theoretic reasons, and since polynomials plotted on a logarithmic scale are logarithmic. This is not a rigorous proof of time complexity, but it gives some guidance. In contrast, the average time complexity for Lazy-Um-3 seems larger than polynomial. In Section 5.3 we will see an analytical argument that the time complexity observed is polynomial, specifically $O(n^4)$.

The average time plots in Figures 5.4 and 5.5 are rather misleading because there is a variation in the time taken to solve different instances, as can be seen in Figure 5.6. Lazy-Um-3 has a very high average time, but it is competitive with Matrix-Um-3, Custom-Um-3 and Toolkit-Um-3 in a large number of cases. However when it does badly, it does very badly and skews the overall average. To give a further impression of the huge variation in Lazy-Um-3 see Figure 5.7 on a logarithmic scale: a very few instances take a very long time.

Table 5.3 is the number of random instances out of 50 when constraint x takes longer to solve than constraint y. For example the entry at the intersection of row 12 and C>M is the number of instances when Custom-Um-3 takes longer than Matrix-Um-3. Table 5.4 does the same for nodes. These reinforce the evidence of the plots, by showing that on specific instances

- Matrix-Um-3 does increasingly well as size increases and is rarely beaten for larger instances;
- Lazy-Um-3 does increasingly badly as size increase; and

[2]The time statistics in this section only include model solving and not building. Build time is proportional to memory usage, and this is described in Section 5.2.2.5.

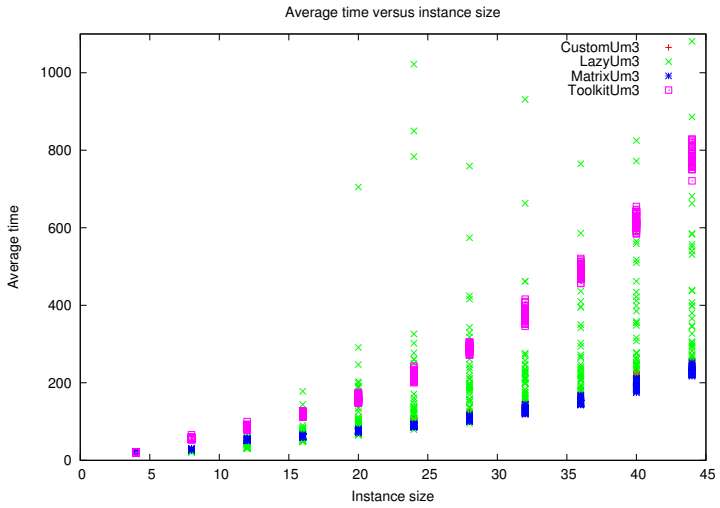


Figure 5.6: Random study: time versus size for static variable order

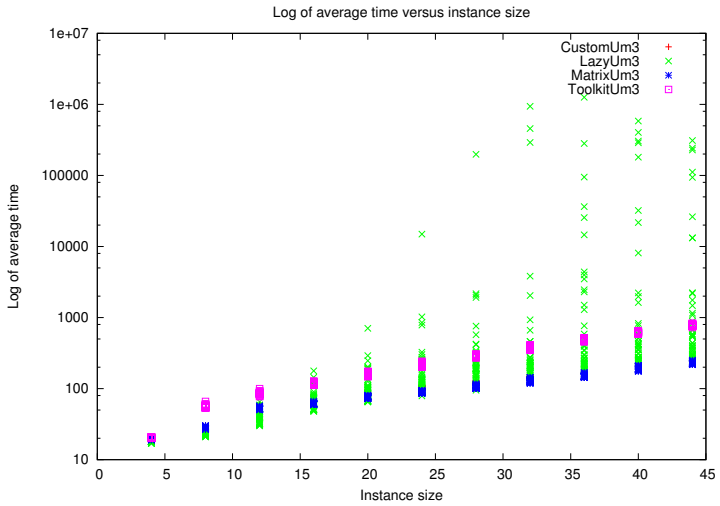


Figure 5.7: Random study: log of time versus size for static variable order

Size	M>C	C>M	M>L	L>M	M>T	T>M	C>L	L>C	C>T	T>C	L>T	T>L
4	6	2	50	0	0	11	50	0	0	13	0	50
8	20	4	50	0	0	50	49	0	0	50	0	50
12	3	39	44	5	0	50	46	3	0	50	0	50
16	0	50	18	30	0	50	28	21	0	50	3	47
20	0	50	11	38	0	50	19	31	0	50	10	40
24	0	50	3	47	0	50	12	37	0	50	10	40
28	0	50	1	49	0	50	6	44	0	50	11	39
32	0	50	0	50	0	50	0	49	0	50	12	38
36	0	50	0	50	0	50	1	49	0	50	16	34
40	0	50	1	49	0	50	1	49	0	50	13	37
44	0	50	0	50	0	50	2	48	0	50	15	34

Table 5.3: Random study: pairwise time comparison with static variable order (M=Matrix-Um-3, C=Custom-Um-3, L=Lazy-Um-3, T=Toolkit-Um-3)

Size	M>C	C>M	M>L	L>M	M>T	T>M	C>L	L>C	C>T	T>C	L>T	T>L
4	0	0	0	15	0	0	0	15	0	0	15	0
8	0	0	0	50	0	0	0	50	0	0	50	0
12	0	0	0	50	0	0	0	50	0	0	50	0
16	0	0	0	50	0	0	0	50	0	0	50	0
20	0	0	0	50	0	0	0	50	0	0	50	0
24	0	0	0	50	0	0	0	50	0	0	50	0
28	0	0	0	50	0	0	0	50	0	0	50	0
32	0	0	0	50	0	0	0	50	0	0	50	0
36	0	0	0	50	0	0	0	50	0	0	50	0
40	0	0	0	50	0	0	0	50	0	0	50	0
44	0	0	0	50	0	0	0	50	0	0	50	0

Table 5.4: Random study: pairwise nodes comparison with static variable order

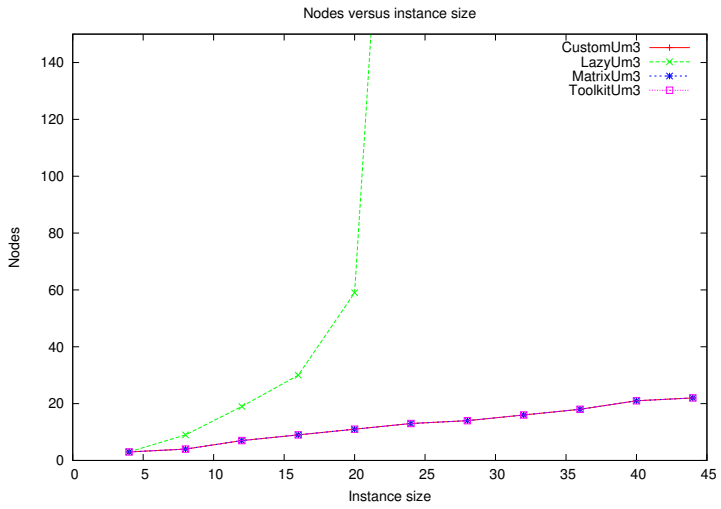


Figure 5.8: Random study: nodes versus size for static variable order

- Custom-Um-3 and Toolkit-Um-3 have similar properties but are slower.

The number of nodes needed (see Figure 5.8) is the same for Custom-Um-3 and Matrix-Um-3, which is exactly as expected, because they use the same propagation algorithm implemented in two different ways! More surprisingly, Toolkit-Um-3 always uses the same number of nodes as these. This is a warning that the random instances are not suitable for showing up the known behavioural difference (Section 3.7) between Toolkit-Um-3 and Custom-Um-3. We will discuss in Section 5.2.3 how an improved trial might address this problem.

Finally, the nodes plot is the same shape as the time plot, because during the search process time is spent at nodes! A plot of the number of nodes for each individual instance has been provided on the CD, but it has the same average and spread properties as Figure 5.6. For a similar reason, a log plot of individual node datums has been left out.

5.2.2.7 Experiments with min-domain dynamic ordering

Figure 5.9 shows that when the min-domain heuristic is applied, the hitherto useless Lazy-Um-3 constraint is the best on average! Furthermore, the variation (illustrated by Figure 5.10) that was previous extremely high is now very low. It seems that the heuristic chooses to instantiate in such a way that Lazy-Um-3 is able to do a lot more propagation than before. The effect on Custom-Um-3, Matrix-Um-3 and Toolkit-Um-3 is a small reduction in average time and a reduction in variance.

These results for solve time suggest a difference in the number of nodes needed by Lazy-Um-3 in the presence of min-domain. In fact, when the min-domain heuristic

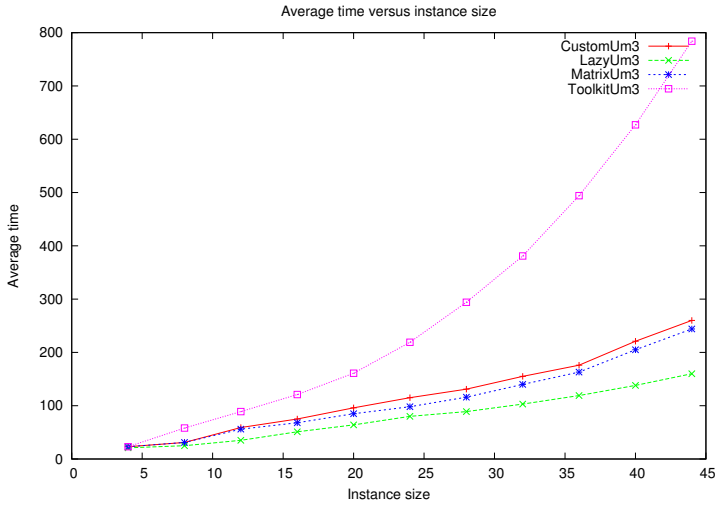


Figure 5.9: Random study: average time versus size for min-domain variable order

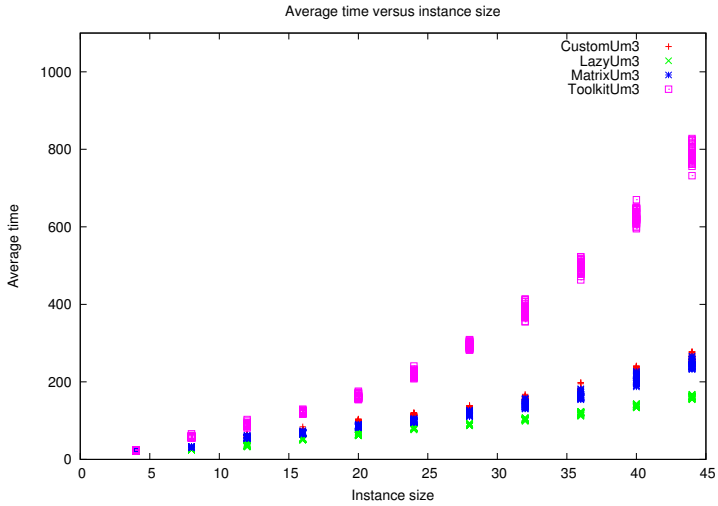


Figure 5.10: Random study: time versus size for min-domain variable order

Size	M>C	C>M	M>L	L>M	M>T	T>M	C>L	L>C	C>T	T>C	L>T	T>L
4	0	0	50	0	0	23	50	0	0	23	0	50
8	12	4	50	0	0	50	50	0	0	50	0	50
12	2	46	50	0	0	50	50	0	0	50	0	50
16	0	50	50	0	0	50	50	0	0	50	0	50
20	0	50	50	0	0	50	50	0	0	50	0	50
24	0	50	50	0	0	50	50	0	0	50	0	50
28	0	50	50	0	0	50	50	0	0	50	0	50
32	0	50	50	0	0	50	50	0	0	50	0	50
36	0	50	50	0	0	50	50	0	0	50	0	50
40	0	50	50	0	0	50	50	0	0	50	0	50
44	0	50	50	0	0	50	50	0	0	50	0	50

Table 5.5: Random study: pairwise time comparison with min-domain variable order

Size	M>C	C>M	M>L	L>M	M>T	T>M	C>L	L>C	C>T	T>C	L>T	T>L
4	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0	0
44	0	0	0	0	0	0	0	0	0	0	0	0

Table 5.6: Random study: pairwise nodes comparison with min-domain variable order

is used the number of nodes used in the search process is *exactly the same* in every generated instance! Tables 5.5 and 5.6 confirm that no constraint runs faster than Lazy-Um-3 on any instance and that every constraint uses exactly the same number of nodes for all instances.

5.2.3 Conclusion

This random study raises several interesting questions as well as providing useful confirmation of some theoretical points.

Model size and speed These results confirm that Custom-Um-3 and Matrix-Um-3 are an enormous improvement over Toolkit-Um-3.

Effect of heuristics and instance difficulty The min-domain variable ordering heuristic has been observed making a large difference compared with a static ordering. Although min-domain was also used for the seabird trials, it did not make Lazy-Um-3 competitive with the others. Does the fact that these random instances are qualitatively "easy" mean that Lazy-Um-3 is able to do well on them? Future work should involve generating random instances of the supertree problems that span the spectrum of difficulties, so that we might know under what circumstances good and bad behaviour occurs.

5.3 Does an AC algorithm solve the whole problem without search?

The results in Sections 5.1 and 5.2 show that using Custom-Um-3 and Matrix-Um-3 constraints, the species model is never observed to use more than one node per variable. This suggests that very little, if any, search is needed. Since search begins with an arc consistent problem, this suggests that in such a consistency level the variables are close

to being a solution already. Experiments have shown that a valid species tree can be obtained by taking as final result the lower bound of each variable from an AC problem. For example:

$$\begin{bmatrix} 0 & 2 & 1 & 1 \\ & 0 & 1 & 1 \\ & & 0 & 1 \\ & & & 0 \end{bmatrix}$$

is a solution obtained from the AC problem

$$\begin{bmatrix} 0 & \{2,3\} & \{1,2\} & \{1,2\} \\ & 0 & \{1,2,3\} & \{1,2,3\} \\ & & 0 & \{1,2,3\} \\ & & & 0 \end{bmatrix}$$

This result has come too late in the project for a thorough investigation, but we will now prove informally that the technique should work and provide a revised worst case time complexity based on it.

The earlier Lemma 6 shows that lower bounds in the ultrametric relation are mutually supportive, meaning that they together conform to the ultrametric relation. Hence the lower bounds of the matrix together form an ultrametric matrix. We mustn't forget, however, that disjunction breaking constraints are also posted to represent triples and fans, and that these constraints are less well understood. Proving that lower bounds support one another in this constraint would involve making sure that an analog of Lemma 6 applies to it. These facts together would prove that AC is indeed sufficient to solve the problem.

A corollary of the fact would be that the time complexity of finding a solution has been reduced to $O(n^4)$. The AC-5 algorithm for making a problem AC is described in Section 1.3.2.2. To make the species tree problem AC, all of the $O(n^3)$ constraints are added to the queue at first; subsequently each constraint may be re-queued up to $O(n^3) \times 3(n-1) = O(n^4)$ times, since each of the $O(n^3)$ constraints can be requeued at most $3(n-1)$ times if propagation only removes one value at a time. $O(1)$ work is done each time a constraint is dequeued, because dequeuing and propagation are $O(1)$, and the $O(1)$ cost of queueing the constraint in the first place can be charged to the dequeueing code. Hence the overall complexity is $(O(n^3) + O(n^4))O(1) = O(n^4)$, this is a significant improvement over the $O(n^{n^2})$ time bound from Section 2.5.4.

Though it seems improbable on the face of it
You must master the huge retards and have faith in the slow
Blossoming of haystacks, stairways, walls of convolvulus,
Until the moon can do no more. Exhausted,
You get out of bed. Your project is completed
Though the experiment is a mess.

— John Ashbery

Chapter 6

Conclusion

> *"Defeat doesnt finish a man—quit does. A man is not finished when he's defeated. He's finished when he quits."*— **Richard M. Nixon**

This project began with a review of the fields of constraint programming and species trees. This was undertaken to provide the necessary background knowledge for a description of two constraint programming solutions to the supertree problem. These solutions suffered from slow performance and large memory requirements. Since the memory usage of these CP solutions was dominated by the $O(n^3)$ space needed for ultrametric constraints, it was hypothesised that reducing memory requirements for this constraint would improve matters overall.

To this end, 4 different constraint designs for the ultrametric constraint were produced and implemented. The Toolkit-Um-3 constraint was a naive encoding of the definition of ultrametric, this was the benchmark against which other implementations were to be tested. The Custom-Um-3 constraint is the centrepiece of the whole project. It maintains generalised bounds arc consistency and has a very simple definition which is the result of exhaustive analysis. A rigorous proof of correctness for the constraint was undertaken and one of the constituent Lemmas leads to a strong hypothesis that arc consistency is sufficient to solve the supertree problem without search in $O(n^4)$ time. This is about a factor of n slower than the best imperative solution. An experimental Lazy-Um-3 constraint was produced to see what effect propagation levels have on the search process. Finally a whole-matrix ultrametric constraint Matrix-Um-3 based on this design was created and this provides an asymptotic improvement in space usage.

Next, a generalised ultrametric constraint called Um-4 was investigated. It is used in a constraint model for the supertree problem that incorporates ancestral divergence dates. A meta-algorithm Custom-Um-4 was used to implement an easily understood and effective constraint for the problem. The above whole-matrix optimisation was applied to create the Matrix-Um-4 constraint providing a further improvement in model memory size.

Finally an empirical study involving both real-life seabird data and random instances was carried out. This study confirms that the new constraints are highly effective in achieving the project's aims.

All in all, this project has been more wide-ranging than was originally envisaged, because the scope has widened to include investigations into the species model itself as well as the constraints it incorporates.

6.1 Future work

The most interesting future direction for research is to find out more about the behaviour of the whole model. We have a strong hypothesis that search is not needed to solve the [GPSW03] supertree model, a next step would be to prove this. Even if this was the case, the CP literature shows (e.g. [Wal93]) that the worst case performance of an algorithm is not always a good guide for its true performance. It would be interesting to carry out another random experiment where the hardness of the problem was controllable, in an attempt to discover when good and bad behaviours occur and to better predict behaviour. Perhaps this would explain the fact that the weak-propagating Lazy-Um-3 constraint does very well on occasions. Many combinatorial problems exhibit phase transition behaviour where hard problems occur on the boundary between easy solvable and easy unsolvable, this may be the case for the supertree problem.

It would also be interesting to undertake a thorough study of variable and value ordering heuristics; other levels of consistency; and other AI approaches to the supertree problem, in the hope of cross-fertilisation.

Appendix A

TOOLKIT-UM-3 propagation

Script `prim_prop_test/cases.sh` contains an example CSP exhibiting every class of starting lower and upper bounds based on Figures 3.2 and 3.3. By supplying "prim" as an argument to this script each problem is made arc consistent using Toolkit-Um-3. Any other argument causes Custom-Um-3 to be used.

Inspection of the output reveals that with Toolkit-Um-3 lower bounds are *never* trimmed although upper bounds are always trimmed correctly. Hence Toolkit-Um-3 maintains a consistency level strictly lower than GBAC.

For example, the following Figures A.1 and A.2 show the same problem being made GBAC by Toolkit-Um-3 and then Custom-Um-3; the lower bounds are ignored by Toolkit-Um-3.

```
1  nmoore@evo:/mnt/cdrom/prim_prop_test$$ java -cp ..:. Test 1 3 2 3 3 3 prim
2  Pb[3 vars, 1 cons]
3  Pb[3 vars, 1 cons]
4  ==== VARIABLES ====
5  x:?[1, 3]
6  y:?[2, 3]
7  z:3[3, 3]
8  Pb[3 vars, 1 cons]
9  ==== CONSTRAINTS ====
10  ( (x:? = y:?) and (z:3 >= y:? + 1) and (z:3 >= x:? + 1) ) or ( (x:? = z:3) and (
       y:? >= z:3 + 1) and (y:? >= x:? + 1) ) or ( (y:? = z:3) and (x:? >= z:3 + 1)
        and (x:? >= y:? + 1) ) or ( (x:? = y:?) and (y:? = z:3) and (x:? = z:3) )
11
12 Pb[3 vars, 1 cons]
13 Pb[3 vars, 1 cons]
14 ==== VARIABLES ====
15 x:?[1, 3]
16 y:?[2, 3]
17 z:3[3, 3]
18 Pb[3 vars, 1 cons]
19 ==== CONSTRAINTS ====
20  ( (x:? = y:?) and (z:3 >= y:? + 1) and (z:3 >= x:? + 1) ) or ( (x:? = z:3) and (
       y:? >= z:3 + 1) and (y:? >= x:? + 1) ) or ( (y:? = z:3) and (x:? >= z:3 + 1)
        and (x:? >= y:? + 1) ) or ( (x:? = y:?) and (y:? = z:3) and (x:? = z:3) )
```

Listing A.1: Propagation with TOOLKIT-UM-3

```
1  nmoore@evo:/mnt/cdrom/prim_prop_test$$ java -cp ..:. Test 1 3 2 3 3 3 cust
2  Pb[3 vars, 1 cons]
3  Pb[3 vars, 1 cons]
4  ==== VARIABLES ====
5  x:?[1, 3]
6  y:?[2, 3]
7  z:3[3, 3]
8  Pb[3 vars, 1 cons]
9  ==== CONSTRAINTS ====
10 um.threevar.BAC.CustomUm3@ec16a4
11
12 Pb[3 vars, 1 cons]
13 Pb[3 vars, 1 cons]
14 ==== VARIABLES ====
15 x:?[2, 3]
16 y:?[2, 3]
17 z:3[3, 3]
18 Pb[3 vars, 1 cons]
19 ==== CONSTRAINTS ====
20 um.threevar.BAC.CustomUm3@ec16a4
```

Listing A.2: Propagation with CUSTOM-UM-3

Appendix B

An instance of the supertree problem

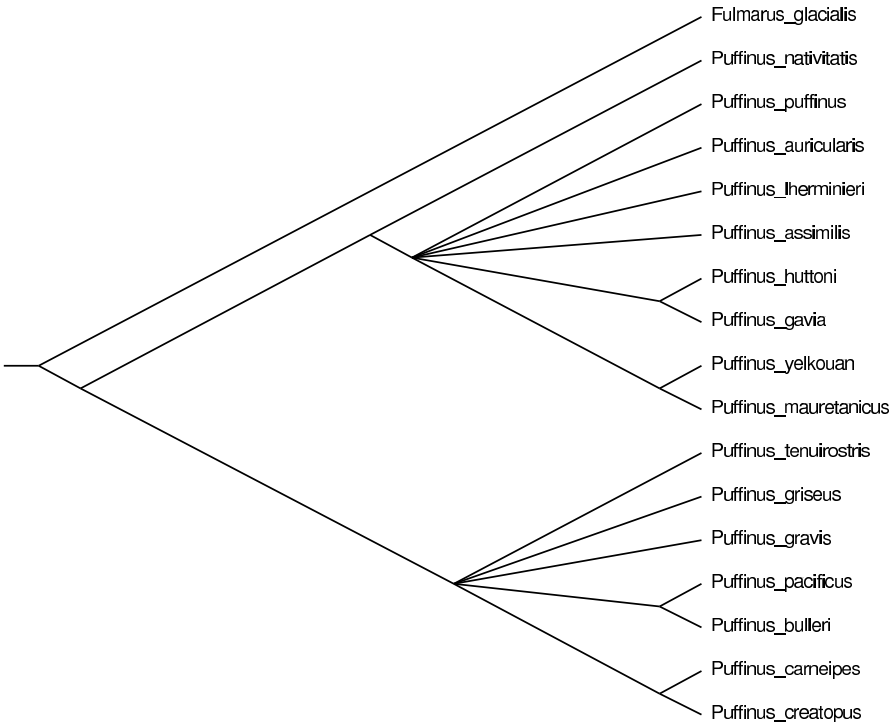


Figure B.1: Seabirds input tree A

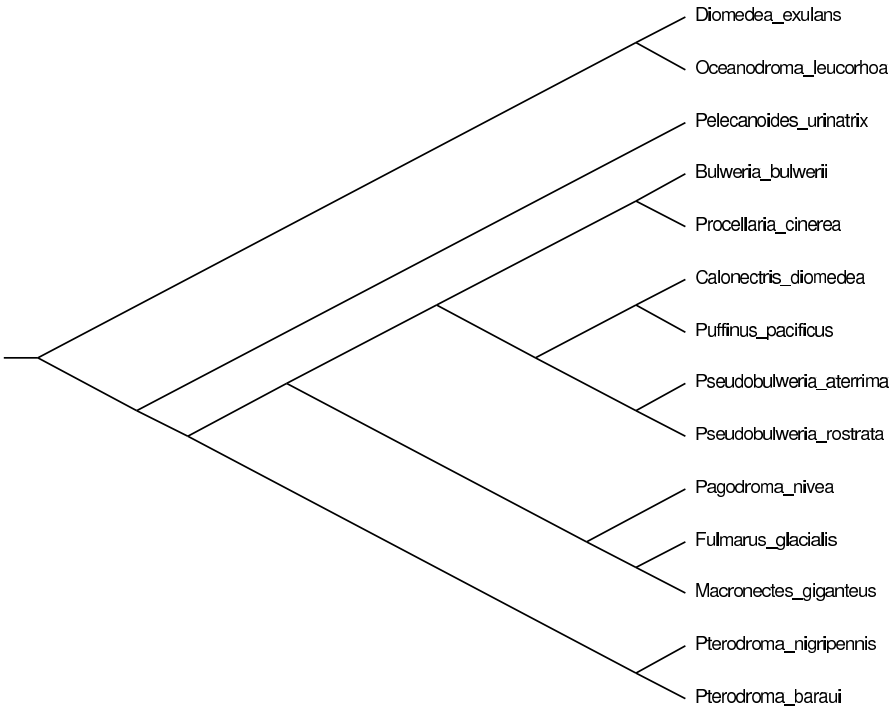


Figure B.2: Seabirds input tree B

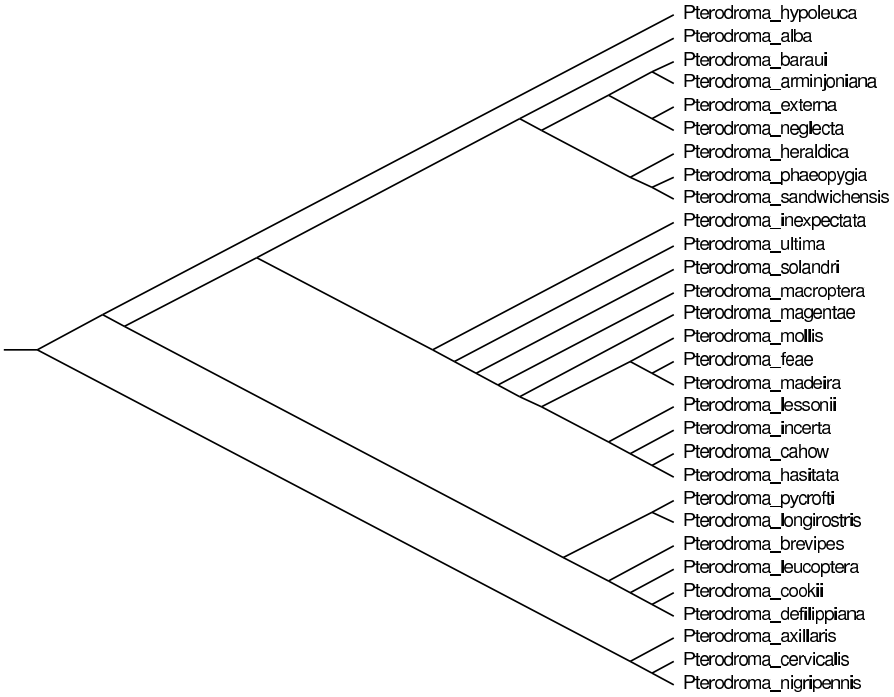


Figure B.3: Seabirds input tree D

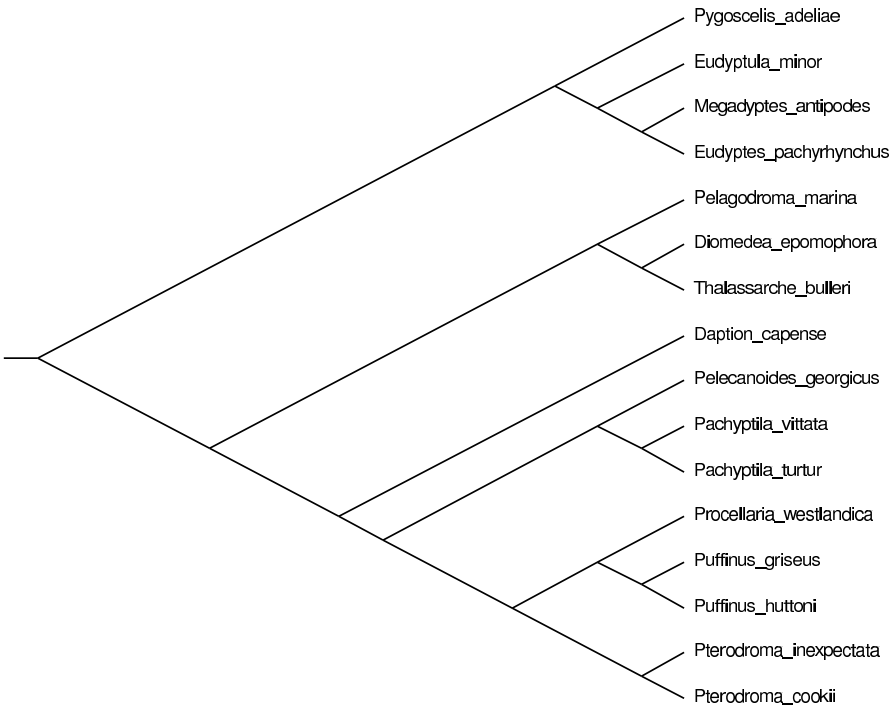


Figure B.4: Seabirds input tree F

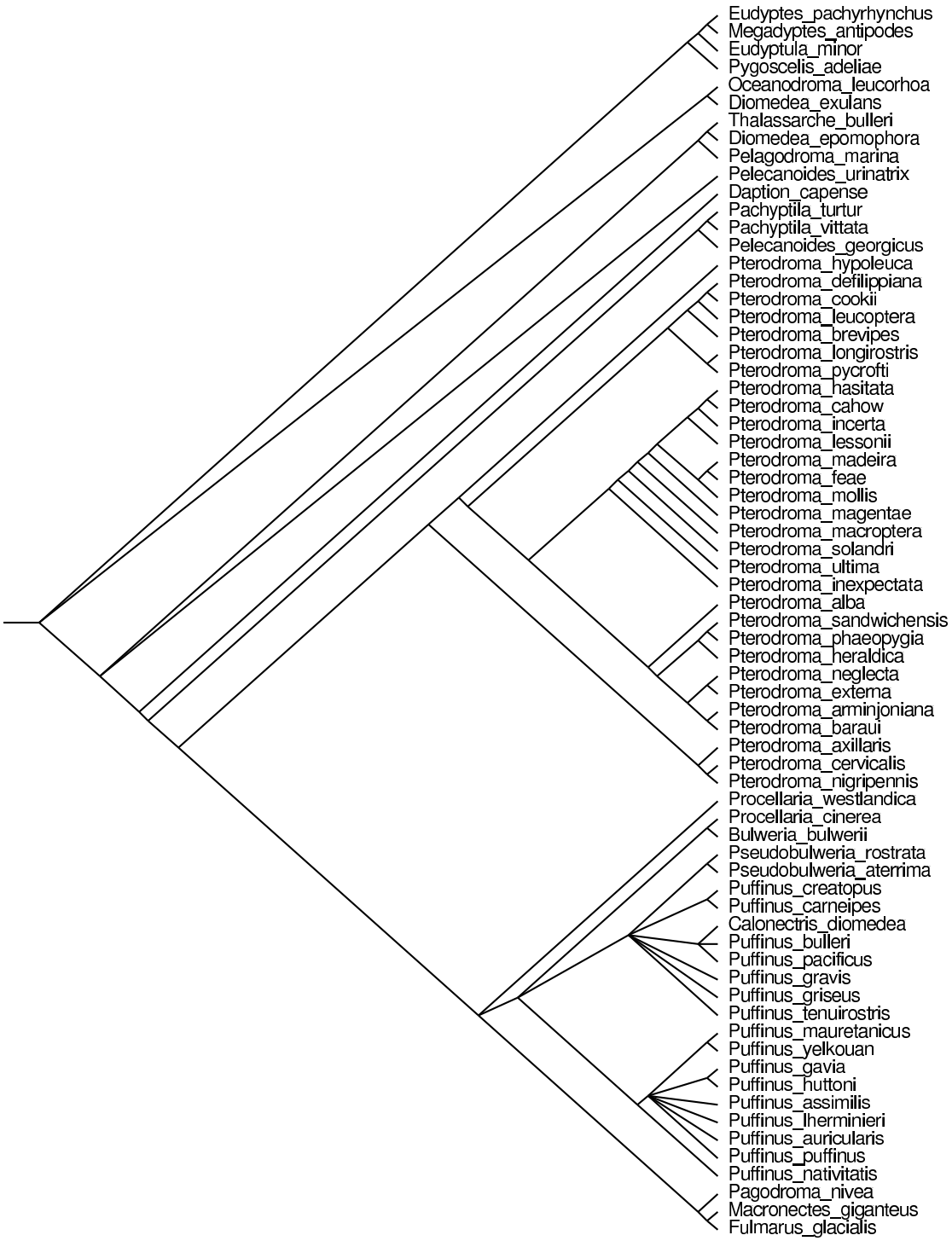


Figure B.5: Supertree ABDF using Custom-Um-3, Matrix-Um-3, Lazy-Um-3, Toolkit-Um-3 and Custom-Um-4 constraints

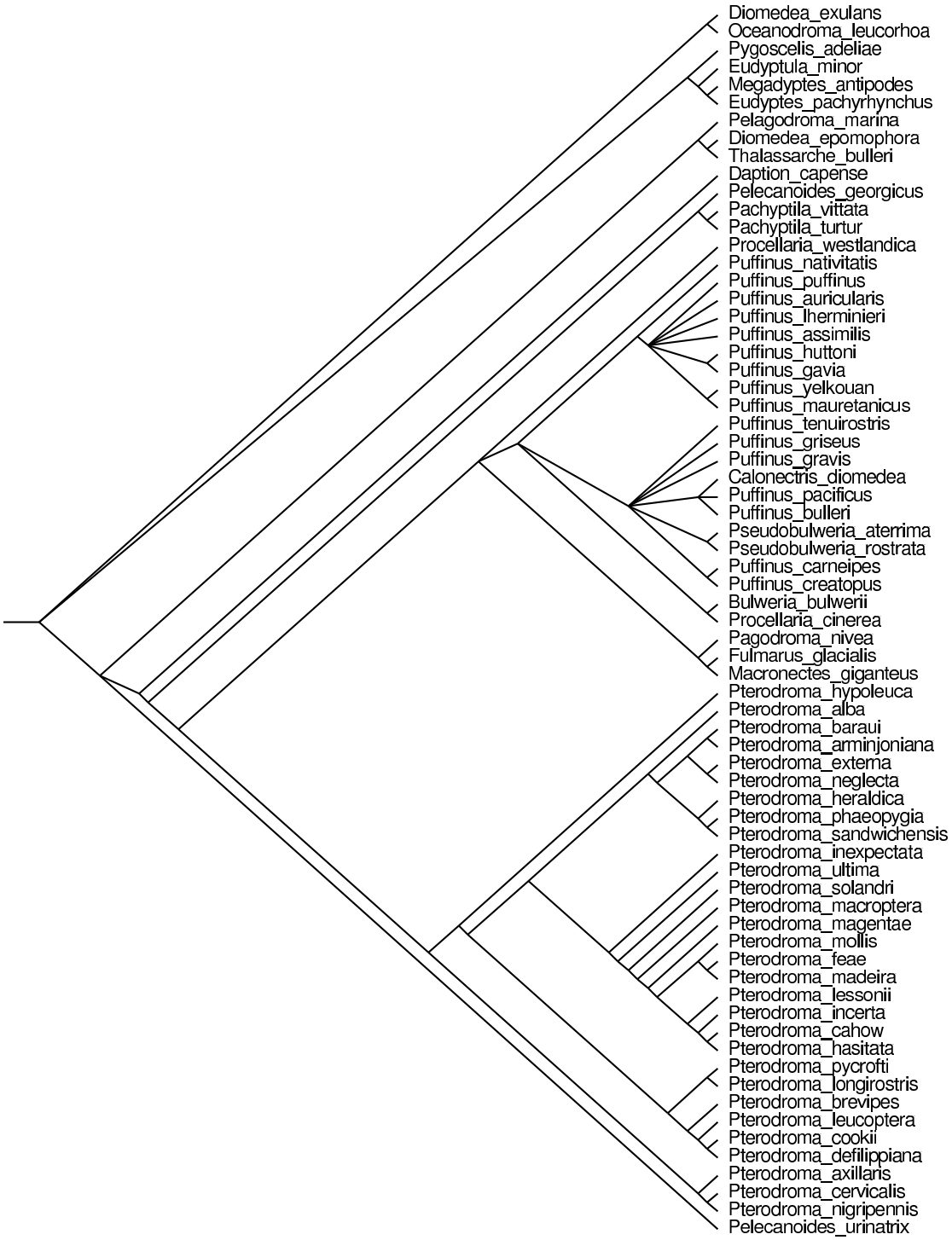


Figure B.6: Supertree ABDF using imperative technique of [NW96]

Appendix C

Remaining code listings

```
1  public class MatrixUm3 extends AbstractLargeIntConstraint {
2      //mapping from variable to matrix index where it came from
3      private Map<IntDomainVar, int[]> varToIndex;
4      //matrix to allow access to constraints by pair of indices
5      private IntDomainVar[][] mat;
6      //size of matrix
7      private int n;
8      //var aliases to allow cut and paste code from CustomUm3.java
9      private IntDomainVar v0;
10     private IntDomainVar v1;
11     private IntDomainVar v2;
12
13     //BEGIN support code copied from CustomUm3.java
14     private class Triple {
15         IntDomainVar s, m, l;
16         public Triple(IntDomainVar s, IntDomainVar m, IntDomainVar l) {
17             this.s = s; this.m = m; this.l = l;
18         }
19     }
20     //return true if and only if v and w's domains have null intersection
21     private static boolean nullIntersection(IntDomainVar v, IntDomainVar w) {
22         return v.getInf() > w.getSup() || v.getSup() < w.getInf();
23     }
24     //return triple of variables ordered by infimum. Uses sorting algorithm for
25     //3 items using at most 3 comparisons, see page 173 of CLR Introduction to
26     //Algorithms.
27     private Triple sortOnInf() {
28         int v0Inf = v0.getInf();
29         int v1Inf = v1.getInf();
30         int v2Inf = v2.getInf();
31         if(v0Inf <= v1Inf)
32             if(v1Inf <= v2Inf)
33                 return new Triple(v0, v1, v2);
34             else
35                 if(v0Inf <= v2Inf)
36                     return new Triple(v0, v2, v1);
37                 else
38                     return new Triple(v2, v0, v1);
39         else
40             if(v0Inf <= v2Inf)
41                 return new Triple(v1, v0, v2);
42             else
43                 if(v1Inf <= v2Inf)
44                     return new Triple(v1, v2, v0);
45                 else
46                     return new Triple(v2, v1, v0);
47     }
48     private Triple sortOnSup() {
49         int v0Sup = v0.getSup();
50         int v1Sup = v1.getSup();
51         int v2Sup = v2.getSup();
52         if(v0Sup <= v1Sup)
53             if(v1Sup <= v2Sup)
54                 return new Triple(v0, v1, v2);
```

```
55              else
56                  if(v0Sup <= v2Sup)
57                      return new Triple(v0, v2, v1);
58                  else
59                      return new Triple(v2, v0, v1);
60          else
61              if(v0Sup <= v2Sup)
62                  return new Triple(v1, v0, v2);
63              else
64                  if(v1Sup <= v2Sup)
65                      return new Triple(v1, v2, v0);
66                  else
67                      return new Triple(v2, v1, v0);
68      }
69      public void fixBounds() throws ContradictionException {
70          if(debug) {
71              System.out.println("fixBounds()");
72              System.out.println("vars: " + v0.pretty() + " " +
73                                 v1.pretty() + " " + v2.pretty());
74          }
75          //FIX UP THE INFS
76          Triple si = sortOnInf();
77          int sInf = si.s.getInf();
78          int mInf = si.m.getInf();
79          int lInf = si.l.getInf();
80          //first case, each inf is different
81          if(sInf != mInf && mInf != lInf) {
82              if(debug) System.out.println("case 1: infs all different:");
83              si.s.setInf(mInf);
84          //4th case, action is identical to first case but temporarily
85          //separated for clarity
86          } else if(mInf == lInf && sInf != mInf) {
87              if(debug) System.out.println("case 4: smallest inf is distinct:");
88              si.s.setInf(mInf);
89          }
90          //FIX UP THE SUPS
91          Triple ss = sortOnSup();
92          int sSup = ss.s.getSup();
93          int mSup = ss.m.getSup();
94          int lSup = ss.l.getSup();
95          //first case, each sup is different
96          if(sSup != mSup && mSup != lSup) {
97              if(debug) System.out.println("case 1: sups all different");
98              if(nullIntersection(ss.l, ss.s)) {
99                  if(debug) System.out.println("null intersection of s and l");
100                 ss.m.setSup(sSup);
101             } else if(nullIntersection(ss.m, ss.s)) {
102                 if(debug) System.out.println("null intersection of m and l");
103                 ss.l.setSup(sSup);
104             }
105         //third case, largest are equal but smallest is different
106         } else if (sSup != mSup && mSup == lSup) {
107             if(debug) System.out.println("case 3: 2 largest sups are equal");
108             if(nullIntersection(ss.m, ss.s)) {
109                 if(debug) System.out.println("null intersection of s and m");
110                 ss.l.setSup(sSup);
111             } else if(nullIntersection(ss.l, ss.s)) {
112                 if(debug) System.out.println("null intersection of s and l");
113                 ss.m.setSup(sSup);
114             }
115         }
116         if(debug) {
117             System.out.println("end vars: " + v0.pretty() + " " +
118                                v1.pretty() + " " + v2.pretty());
119             System.out.println("fixBounds() ends");
120         }
121     }
122     //END support code copied from CustomUm3.java
123
124     public static IntDomainVar[] getUseful(IntDomainVar[][] mat) {
125         int n = mat.length;
126         int numUseful = n * (n - 1) / 2;
127         IntDomainVar[] vars = new IntDomainVar[numUseful];
```

```
128             int count = 0;
129             //extract useful variables
130             for(int i = 0; i < n - 1; i++)
131                 for(int j = i + 1; j < n; j++)
132                     vars[count++] = mat[i][j];
133             return vars;
134         }
135         public MatrixUm3(IntDomainVar[][] mat) {
136             //the superclass constructor must come first so we must flatten the
137             //matrix inside a function call
138             super(getUseful(mat));
139             n = mat.length;
140             int numUseful = n * (n - 1) / 2;
141             varToIndex = new HashMap<IntDomainVar, int[]>(3 * numUseful);
142             //add all useful variables to mapping
143             for(int i = 0; i < n - 1; i++)
144                 for(int j = i + 1; j < n; j++)
145                     varToIndex.put(mat[i][j], new int[] { i, j });
146             this.mat = mat;
147         }
148 
149         private void doPropagate(int idx) throws ContradictionException {
150             v0 = getIntVar(idx);
151             int[] index = varToIndex.get(v0);
152             int i = index[0]; int j = index[1];
153             for(int k = 0; k < n; k++) {
154                 if(i != k && j != k) {
155                     v1 = mat[i][k];
156                     v2 = mat[j][k];
157                     fixBounds();
158                 }
159             }
160         }
161 
162         public void awakeOnInf(int idx) throws ContradictionException {
163             doPropagate(idx);
164         }
165 
166         public void awakeOnSup(int idx) throws ContradictionException {
167             doPropagate(idx);
168         }
169 
170         public void awake() throws ContradictionException {
171             for(int i = 0; i < getNbVars(); i++)
172                 doPropagate(i);
173         }
174 
175         public void awakeOnRem(int idx, int x) throws ContradictionException {
176             ;
177         }
178 
179         public void awakeOnRemovals(int idx, IntIterator deltaDom)
180             throws ContradictionException {
181             ;
182         }
183 
184         public void propagate() throws ContradictionException {
185             ;
186         }
187 
188         public void awakeOnInst(int idx) throws ContradictionException {
189             doPropagate(idx);
190         }
191 
192         public boolean isSatisfied() {
193             return true;
194         }
195     }
```

Listing C.1: Implementation of MATRIX-UM-3

```
1  public class CustomUm4 extends AbstractLargeIntConstraint {
2      //additional references to variables, to avoid indexing <vars> every time
3      public IntDomainVar v0;
4      public IntDomainVar v1;
5      public IntDomainVar v2;
6      public IntDomainVar d;
7
8      public CustomUm4(IntDomainVar v0, IntDomainVar v1, IntDomainVar v2,
9                       IntDomainVar d) {
10         super(new IntDomainVar[] {v0, v1, v2, d});
11         this.v0 = v0;
12         this.v1 = v1;
13         this.v2 = v2;
14         this.d = d;
15
16
17     private static final int max(int a, int b) { return Math.max(a, b); }
18     private static final int min(int a, int b) { return Math.min(a, b); }
19     private static final int max(int a, int b, int c) {
20         return Math.max(a, Math.max(b, c));
21     }
22     private static final int min(int a, int b, int c) {
23         return Math.min(a, Math.min(b, c));
24     }
25     //return true if and only if v and w's domains have null intersection
26     private static boolean nullInter(IntDomainVar v, IntDomainVar w) {
27         return v.getInf() > w.getSup() || v.getSup() < w.getInf();
28     }
29
30     //propagate changes in d to v0, v1 and v2
31     public void fromDToV() throws ContradictionException {
32         int v0Inf = v0.getInf(); int newV0Inf = Integer.MAX_VALUE;
33         int v1Inf = v1.getInf(); int newV1Inf = Integer.MAX_VALUE;
34         int v2Inf = v2.getInf(); int newV2Inf = Integer.MAX_VALUE;
35         int v0Sup = v0.getSup(); int newV0Sup = Integer.MIN_VALUE;
36         int v1Sup = v1.getSup(); int newV1Sup = Integer.MIN_VALUE;
37         int v2Sup = v2.getSup(); int newV2Sup = Integer.MIN_VALUE;
38         if(d.canBeInstantiatedTo(1)) {
39             int expr1 = max(v1Inf, v2Inf);
40             newV0Inf = min(newV0Inf, expr1 + 1);
41             newV1Inf = min(newV1Inf, expr1);
42             newV2Inf = min(newV2Inf, expr1);
43             int expr2 = min(v1Sup, v2Sup, v0Sup - 1);
44             newV0Sup = max(newV0Sup, v0Sup);
45             newV1Sup = max(newV1Sup, expr2);
46             newV2Sup = max(newV2Sup, expr2);
47         }
48         if(d.canBeInstantiatedTo(2)) {
49             int expr1 = max(v0Inf, v2Inf);
50             newV1Inf = min(newV1Inf, expr1 + 1);
51             newV0Inf = min(newV0Inf, expr1);
52             newV2Inf = min(newV2Inf, expr1);
53             int expr2 = min(v0Sup, v2Sup, v1Sup - 1);
54             newV0Sup = max(newV0Sup, expr2);
55             newV1Sup = max(newV1Sup, v1Sup);
56             newV2Sup = max(newV2Sup, expr2);
57         }
58         if(d.canBeInstantiatedTo(3)) {
59             int expr1 = max(v0Inf, v1Inf);
60             newV2Inf = min(newV2Inf, expr1 + 1);
61             newV0Inf = min(newV0Inf, expr1);
62             newV1Inf = min(newV1Inf, expr1);
63             int expr2 = min(v0Sup, v1Sup, v2Sup - 1);
64             newV0Sup = max(newV0Sup, expr2);
65             newV1Sup = max(newV1Sup, expr2);
66             newV2Sup = max(newV2Sup, v2Sup);
67         }
68         if(d.canBeInstantiatedTo(4)) {
69             int minSup = min(v0Sup, v1Sup, v2Sup);
70             int maxInf = max(v0Inf, v1Inf, v2Inf);
71             newV0Inf = min(newV0Inf, maxInf);
72             newV1Inf = min(newV1Inf, maxInf);
73             newV2Inf = min(newV2Inf, maxInf);
```

```
74              newV0Sup = max(newV0Sup, minSup);
75              newV1Sup = max(newV1Sup, minSup);
76              newV2Sup = max(newV2Sup, minSup);
77          }
78          if(debug) {
79              System.out.println("v0.inf<-" + newV0Inf);
80              System.out.println("v1.inf<-" + newV1Inf);
81              System.out.println("v2.inf<-" + newV2Inf);
82              System.out.println("v0.sup<-" + newV0Sup);
83              System.out.println("v1.sup<-" + newV1Sup);
84              System.out.println("v2.sup<-" + newV2Sup);
85          }
86          v0.setInf(newV0Inf); v1.setInf(newV1Inf); v2.setInf(newV2Inf);
87          v0.setSup(newV0Sup); v1.setSup(newV1Sup); v2.setSup(newV2Sup);
88      }
89
90      //propagate changes in v0, v1 and v2 to d
91      public void fromVToD() throws ContradictionException {
92          if(nullInter(v1, v2) || v0.getSup() <= max(v1.getInf(), v2.getInf()))
93              d.remVal(1);
94          if(nullInter(v0, v2) || v1.getSup() <= max(v0.getInf(), v2.getInf()))
95              d.remVal(2);
96          if(nullInter(v0, v1) || v2.getSup() <= max(v0.getInf(), v1.getInf()))
97              d.remVal(3);
98          if(nullInter(v0, v1) || nullInter(v0, v2) || nullInter(v1, v2))
99              d.remVal(4);
100     }
101
102     public void eventDispatch(int idx) throws ContradictionException {
103         fromDToV();
104         if(idx != 3)
105             fromVToD();
106     }
107
108     public void awake() throws ContradictionException {
109         fromVToD();
110         fromDToV();
111     }
112
113     public void awakeOnInf(int idx) throws ContradictionException {
114         eventDispatch(idx);
115     }
116
117     public void awakeOnInst(int idx) throws ContradictionException {
118         eventDispatch(idx);
119     }
120
121     public void awakeOnRem(int idx, int a) throws ContradictionException {
122         eventDispatch(idx);
123     }
124
125     public void awakeOnRemovals(int idx, IntIterator deltaDomain)
126         throws ContradictionException {
127         eventDispatch(idx);
128     }
129
130     public void awakeOnSup(int idx) throws ContradictionException {
131         eventDispatch(idx);
132     }
133
134     public boolean isSatisfied() {
135         return true;
136     }
137 }
```

Listing C.2: Implementation of CUSTOM-UM-4

```
1  public class MatrixUm4 extends AbstractLargeIntConstraint {
2      //record of parameters
3      private IntDomainVar[][] mat;
4      private IntDomainVar[][][] ds;
5      private int n;
6      //aliases for variables to allow easy import from CustomUm4.java
7      private IntDomainVar v0;
8      private IntDomainVar v1;
9      private IntDomainVar v2;
10     private IntDomainVar d;
11     //quick way to find out if a variable is one of the ds, val!=null means yes
12     private Map<IntDomainVar, Object> isD;
13     //unique mapping from d to array of 3 vs
14     private Map<IntDomainVar, IntDomainVar[]> dToVs;
15     //mapping from variable to it's position in the matrix
16     private Map<IntDomainVar, int[]> varToIndex;
17
18     //BEGIN unchanged code import from CustomUm4.java
19     private static final int max(int a, int b) { return Math.max(a, b); }
20     private static final int min(int a, int b) { return Math.min(a, b); }
21     private static final int max(int a, int b, int c) {
22         return Math.max(a, Math.max(b, c));
23     }
24     private static final int min(int a, int b, int c) {
25         return Math.min(a, Math.min(b, c));
26     }
27     //return true if and only if v and w's domains have null intersection
28     private static boolean nullInter(IntDomainVar v, IntDomainVar w) {
29         return v.getInf() > w.getSup() || v.getSup() < w.getInf();
30     }
31     //propagate changes in d to v0, v1 and v2
32     public void fromDToV() throws ContradictionException {
33         if(debug) {
34             System.out.println("fromDToV()");
35             System.out.println(v0.pretty() + "," + v1.pretty() + "," +
36                                v2.pretty() + "," + d.pretty());
37         }
38         int v0Inf = v0.getInf(); int newV0Inf = Integer.MAX_VALUE;
39         int v1Inf = v1.getInf(); int newV1Inf = Integer.MAX_VALUE;
40         int v2Inf = v2.getInf(); int newV2Inf = Integer.MAX_VALUE;
41         int v0Sup = v0.getSup(); int newV0Sup = Integer.MIN_VALUE;
42         int v1Sup = v1.getSup(); int newV1Sup = Integer.MIN_VALUE;
43         int v2Sup = v2.getSup(); int newV2Sup = Integer.MIN_VALUE;
44         if(d.canBeInstantiatedTo(1)) {
45             int expr1 = max(v1Inf, v2Inf);
46             newV0Inf = min(newV0Inf, expr1 + 1);
47             newV1Inf = min(newV1Inf, expr1);
48             newV2Inf = min(newV2Inf, expr1);
49             int expr2 = min(v1Sup, v2Sup, v0Sup - 1);
50             newV0Sup = max(newV0Sup, v0Sup);
51             newV1Sup = max(newV1Sup, expr2);
52             newV2Sup = max(newV2Sup, expr2);
53         }
54         if(d.canBeInstantiatedTo(2)) {
55             int expr1 = max(v0Inf, v2Inf);
56             newV1Inf = min(newV1Inf, expr1 + 1);
57             newV0Inf = min(newV0Inf, expr1);
58             newV2Inf = min(newV2Inf, expr1);
59             int expr2 = min(v0Sup, v2Sup, v1Sup - 1);
60             newV0Sup = max(newV0Sup, expr2);
61             newV1Sup = max(newV1Sup, v1Sup);
62             newV2Sup = max(newV2Sup, expr2);
63         }
64         if(d.canBeInstantiatedTo(3)) {
65             int expr1 = max(v0Inf, v1Inf);
66             newV2Inf = min(newV2Inf, expr1 + 1);
67             newV0Inf = min(newV0Inf, expr1);
68             newV1Inf = min(newV1Inf, expr1);
69             int expr2 = min(v0Sup, v1Sup, v2Sup - 1);
70             newV0Sup = max(newV0Sup, expr2);
71             newV1Sup = max(newV1Sup, expr2);
72             newV2Sup = max(newV2Sup, v2Sup);
73         }
```

```
74          if(d.canBeInstantiatedTo(4)) {
75              int minSup = min(v0Sup, v1Sup, v2Sup);
76              int maxInf = max(v0Inf, v1Inf, v2Inf);
77              newV0Inf = min(newV0Inf, maxInf);
78              newV1Inf = min(newV1Inf, maxInf);
79              newV2Inf = min(newV2Inf, maxInf);
80              newV0Sup = max(newV0Sup, minSup);
81              newV1Sup = max(newV1Sup, minSup);
82              newV2Sup = max(newV2Sup, minSup);
83          }
84          if(debug) {
85              System.out.println("v0.inf<-" + newV0Inf);
86              System.out.println("v1.inf<-" + newV1Inf);
87              System.out.println("v2.inf<-" + newV2Inf);
88              System.out.println("v0.sup<-" + newV0Sup);
89              System.out.println("v1.sup<-" + newV1Sup);
90              System.out.println("v2.sup<-" + newV2Sup);
91          }
92          v0.setInf(newV0Inf); v1.setInf(newV1Inf); v2.setInf(newV2Inf);
93          v0.setSup(newV0Sup); v1.setSup(newV1Sup); v2.setSup(newV2Sup);
94          if(debug) System.out.println("fromDToV()␣ends");
95      }
96
97      //propagate changes in v0, v1 and v2 to d
98      public void fromVToD() throws ContradictionException {
99          if(debug) {
100             System.out.println("fromVToD()");
101             System.out.println(v0.pretty() + "," + v1.pretty() + "," +
102                                v2.pretty() + "," + d.pretty());
103         }
104         if(nullInter(v1, v2) || v0.getSup() <= max(v1.getInf(), v2.getInf())) {
105             if(debug) System.out.println("d␣losing␣1");
106             d.remVal(1);
107         }
108         if(nullInter(v0, v2) || v1.getSup() <= max(v0.getInf(), v2.getInf())) {
109             if(debug) System.out.println("d␣losing␣2");
110             d.remVal(2);
111         }
112         if(nullInter(v0, v1) || v2.getSup() <= max(v0.getInf(), v1.getInf())) {
113             d.remVal(3);
114             if(debug) System.out.println("d␣losing␣3");
115         }
116         if(nullInter(v0, v1) || nullInter(v0, v2) || nullInter(v1, v2)) {
117             d.remVal(4);
118             if(debug) System.out.println("d␣losing␣4");
119         }
120         if(debug) System.out.println("fromVToD()␣ends");
121     }
122     //END unchanged code import from CustomUm4.java
123
124     //get unique variables from the matrix and unique d variables
125     public static IntDomainVar[] getUsefulVars(IntDomainVar[][] mat,
126                                                IntDomainVar[][][] ds) {
127         int n = mat.length;
128         int uniqueVs = n * (n - 1) / 2;
129         int uniqueDs = n * (n - 1) * (n - 2) / 6; //C(n,3)
130         IntDomainVar[] vars = new IntDomainVar[uniqueVs + uniqueDs];
131         int count = 0;
132         for(int i = 0; i < n - 1; i++)
133             for(int j = i + 1; j < n; j++)
134                 vars[count++] = mat[i][j];
135         for(int i = 0; i < n - 2; i++)
136             for(int j = i + 1; j < n - 1; j++)
137                 for(int k = j + 1; k < n; k++)
138                     vars[count++] = ds[i][j][k];
139         return vars;
140     }
141     public MatrixUm4(IntDomainVar[][] mat, IntDomainVar[][][] ds) {
142         super(getUsefulVars(mat, ds));
143         n = mat.length;
144         this.mat = mat;
145         this.ds = ds;
146         int usefulDs = n * (n-1) * (n-2) / 6;
```

```
147         int usefulVs = n * (n-1) / 2;
148         isD = new HashMap<IntDomainVar, Object>(3 * usefulDs);
149         varToIndex = new HashMap<IntDomainVar, int[]>(3 * usefulVs);
150         dToVs = new HashMap<IntDomainVar, IntDomainVar[]>(3 * usefulDs);
151         for(int i = 0; i < n - 1; i++)
152             for(int j = i + 1; j < n; j++)
153                 varToIndex.put(mat[i][j], new int[] {i, j});
154         for(int i = 0; i < n - 2; i++) {
155             for(int j = i + 1; j < n - 1; j++) {
156                 for(int k = j + 1; k < n; k++) {
157                     isD.put(ds[i][j][k], true);
158                     dToVs.put(ds[i][j][k],
159                               new IntDomainVar[] {mat[i][j],
160                                                   mat[i][k],
161                                                   mat[j][k]});
162                 }
163             }
164         }
165     }
166
167     //sort 3 values using fixed sorting procedure of no more than 3 comparisons.
168     private int[] sortThree(int x, int y, int z) {
169         if(x <= y)
170             if(y <= z)
171                 return new int[] {x, y, z};
172             else
173                 if(x <= z)
174                     return new int[] {x, z, y};
175                 else
176                     return new int[] {z, x, y};
177         else
178             if(x <= z)
179                 return new int[] {y, x, z};
180             else
181                 if(y <= z)
182                     return new int[] {y, z, x};
183                 else
184                     return new int[] {z, y, x};
185     }
186     public void eventDispatch(int idx) throws ContradictionException {
187         IntDomainVar v = getIntVar(idx);
188         if(isD.get(v) != null) {
189             d = v;
190             IntDomainVar[] vs = dToVs.get(d);
191             v0 = vs[0]; v1 = vs[1]; v2 = vs[2];
192             fromDToV();
193         } else {
194             int[] ij = varToIndex.get(v);
195             int i = ij[0]; int j = ij[1];
196             for(int k = 0; k < n; k++) {
197                 if(k != i && k != j) {
198                     //it's important that the order of v0,v1,v2 is always the
199                     //same whenever they're the same variables, because d is
200                     //dependent on the order
201                     int[] sort = sortThree(i, j, k);
202                     v0 = mat[sort[0]][sort[1]];
203                     v1 = mat[sort[0]][sort[2]];
204                     v2 = mat[sort[1]][sort[2]];
205                     d = ds[sort[0]][sort[1]][sort[2]];
206                     fromDToV();
207                     fromVToD();
208                 }
209             }
210         }
211     }
212
213     public void awake() throws ContradictionException {
214         for(int i = 0; i < n - 2; i++) {
215             for(int j = i + 1; j < n - 1; j++) {
216                 for(int k = j + 1; k < n; k++) {
217                     d = ds[i][j][k];
218                     v0 = mat[i][j];
219                     v1 = mat[i][k];
```

```
220                     v2 = mat[j][k];
221                     fromDToV();
222                     fromVToD();
223                 }
224             }
225         }
226     }
227
228     public void awakeOnInf(int idx) throws ContradictionException {
229         eventDispatch(idx);
230     }
231
232     public void awakeOnInst(int idx) throws ContradictionException {
233         eventDispatch(idx);
234     }
235
236     public void awakeOnRem(int idx, int a) throws ContradictionException {
237         eventDispatch(idx);
238     }
239
240     public void awakeOnRemovals(int idx, IntIterator deltaDomain)
241         throws ContradictionException {
242         eventDispatch(idx);
243     }
244
245     public void awakeOnSup(int idx) throws ContradictionException {
246         eventDispatch(idx);
247     }
248
249     public boolean isSatisfied() {
250         return true;
251     }
252 }
```

Listing C.3: Implementation of Matrix-Um-4

Appendix D

Electronic materials

The CDROM contains code for all the constraints and models used in the project, as well as sample constraint programs, scripts for running experiments, data used in experiments, this report, JChoco source and API.

`api/*` The JChoco 1.1.04 API in HTML format.

`birds/*` Birds species data in Newick format originating from [KP02].

`choco/*` The JChoco 1.1.04 library as Java source and compiled classes.

`empirical` Data, scripts and constraint models used in the empirical study and described elsewhere.

`empirical/birds/*` Seabird data.

`empirical/datares_dyn` Random data and results for min-domain variable order test.

`empirical/datares_stat` Random data and results for static variable order test.

`empirical/model` JChoco model to do supertree problem

`empirical/RandomBTree.java` Java code to generate random bifurcating tree.

`empirical/trees_constr_model` JChoco model to do supertree problem but outputting more data.

`empirical/trees_conv_model` Java imperative solution to supertree problem, used by `birdsrun.sh`

`practice` Practice programs written while learning CP.

`practice/cp4_ex_1` Crystal maze puzzle solution.

`practice/less_than_constraint` Code for $<$ constraint.

`practice/pin_num` PIN puzzle solution.

`practice/zebra` CP solution to the well known "zebra puzzle"

`presentation` LaTeX beamer presentation.

`prim_prop_test` Scripts used to test levels of propgation for Custom-Um-3 and Toolkit-Um-3.

`proposal.txt` Project proposal written in September.

`report` LaTeX sources and graphics for this report.

`supertreesNew` Prosser's JChoco supertree models.

`um` Java library containing implementations of the new constraints.

`um/fourvar/Test.java` Test program for Um-4 constraints.

`um/fourvar/buildin` Location of Toolkit-Um-4.

`um/fourvar/custom` Location of Custom-Um-4.

`um/matrix/Test*.java` Test programs for matrix constraints.

`um/matrix/AC3BAC` Location of Matrix-Um-3.

`um/matrix/AC3BAC4` Location of Matrix-Um-4.

`um/threevar/Test.java` Test program for Um-3 constraints.

`um/threevar/BAC` Location of Custom-Um-3.

`um/threevar/builtin` Location of Toolkit-Um-3.

`um/threevar/wait2` Location of Lazy-Um-3.

D.1 Running the code

The code requires a Java virtual machine supporting the language version 1.5 (including generics). The code in the project is organised as a package called `um` which must be in the Java classpath. The JChoco package also needs to be in the classpath. For example, to run a supertree model do the following:

```
1 nmoore@evo:/mnt/cdrom/SUPERTREESMODEL$$ java -cp /mnt/cdrom:. Build \
2                                             /mnt/cdrom/birds/birdsA.tre \
3                                             /mnt/cdrom/birds/birdsB.tre
```

Listing D.1: Example invocation of the supertree model in BASH

D.1.1 Using code in other models

The code is organised as a JAVA package called `um`. The layout is shown above under the directory `um`. To use the code it must be correctly `import`ed and the classpath for the JAVA VM must include the package.

Appendix E

Experimental data

E.1 Experimental setup

The experiments were run on a Compaq Evo N800v laptop with a 1.8GHz Pentium IV processor and 768Mb of RAM, running Ubuntu GNU/Linux with kernel version 2.6.15-28. Sun Java build 1.5.0_06-b05 was used to run the programs.

LINUX may choose to use virtual memory to store JAVA objects such as those that comprise a JCHOCO model; if this is done then the search process may be considerably slower. For increasingly large programs, this effect will result in a crossing over point when results will suddenly slow down as virtual memory begins to be used, resulting in an illusory change in execution time. To remove this potential source of variability the tests were run with virtual memory disabled (using the `swapoff` utility).

E.2 Sources of statistics

The required performance data is obtained from the following sources:

time The difference in LINUX's real time clock before and after the event being timed.

nodes JCHOCO provides this statistic.

memory usage The difference between the space occupied in JAVA's heap before and after the model is loaded.

The fact that garbage collection in JAVA is potentially unpredictable means that the readings of occupied heap space could be misleading. For example if a "before" reading includes garbage but an "after" reading does not, then the difference is not the size of the model but the size of the model minus the amount of garbage collected! The instrumentation code attempts to avoid this problem by forcing JAVA to garbage collect before each sample is taken, though this is not a perfect solution because JAVA is at liberty to ignore the request[Mic07].

JAVA's heap is variable by default, and this can affect readings of occupied space. For this reason the heap allocated to JAVA was fixed at 629MB[1] for all experiments.

E.3 Running the random empirical study

The scripts described in this Section expect a directory called `datares` with subdirectories `filtered`, `results`, `run` and `trees`. These directories should be empty before the

[1]The memory size argument to the `java` executable is "600M", but this means $600 \times 1024 \times 1024$ bytes. Since a MB is really 10^6 bytes[Var07a] "600M" amounts to 629.15MB.

scripts are run, because the scripts are not idempotent. The following is a sequence of steps required to repeat the experiments:

1. Run `gentrees.sh` to create a set of random trees. First argument is the number of each size required; the rest of the arguments are the sizes required. They are stored in `datares/trees`.

2. Run `doruns.sh` to run the model on data files. The first argument is the number of repetitions to do on each instance; the remaining arguments are the sizes of files to be used. The results are stored in `datares/run`.

3. Run `medianFilter.sh` to median filter time statistics for each size and constraint. First argument is number of repetitions to average over; second argument is number of instances there are of each size; remaining arguments are the sizes to be used. Results stored in `datares/filtered`.

4. Run scripts `resGather*.sh` to produce data files suitable for plotting. Results go in `datares/results`.

5. Execute `gnuplot f` for each file f whose name ends with "plot", to produce graphs of all the data in `datares`.

Appendix F

Original project proposal

The motivation for the project is the combination of leaf labelled rooted trees sharing labels, such as those found in a collection of overlapping phylogenetic trees. A constraint programming solution has been produced in JChoco by representing species trees (which are necessarily ultrametric) as the corresponding ultrametric matrix. The current method of constraining the matrix to be ultrametric uses JChocotoolkit primitives and this puts a limit on the size of problems that can be modelled.

A first indended outcome of this project is to produce a specialised constraint to ensure that $\mathrm{d}(x, y) \leq \mathrm{d}(x, z) = \mathrm{d}(y, z)$ or $\mathrm{d}(x, y) = \mathrm{d}(x, z) = \mathrm{d}(y, z)$ where x, y, z are one each of the 3 constrained integer variables. This satisfies one of the main conditions on ultrametricity.

A second intended outcome is to address the fact that a cubic number of constraints are needed to ensure a matrix is ultrametric. This could be tackled by the production of a single "ultrametric" constraint over a matrix.

These solutions should be tested and possibly proved correct before being incorporated into existing code for more testing, benchmarking and use.

Further extension work such as explanations of the tree produced and the discovery of the "core" tree over all solutions is possible.

Before beginning the above work it will be necessary to gain a familiarity with constraint programming and the internals of the JChoco toolkit and to gain a reasonable understanding of the problem domain of phylogenetic trees and tree construction. The DCS course on Constraint Programming will not commence until second semester so this course comes too late to help significantly.

Appendix G

Project log

> *"If I had been good at making estimates of how long something was going to take, I never would have started."*— **Donald E. Knuth**

11th September 2006 Meet with Patrick to discuss project and wrote project proposal based on this. Briefly reviewed literature in the area of evolution, phylogenetic trees and supertrees.

18th September 2006 Read Barbara Smith's introduction to CP. Read JChocomanual. Modelled zebra and pin number puzzles in JChoco. Explored alternative modellings and, informally, levels of propagation.

25th September 2006 Derived formula to calculate the number of bifurcating trees up to symmetry. This came in very useful in generating random trees in the empirical study.

2nd October 2006 Modelled crystal maze puzzle in JChoco. Wrote background section on evolution and species trees. Read AC-5 paper to formalise understanding of propagation. Designed a $<$ (less than) constraint and coded it in JChoco.

9th October 2006 Designed GAC Um-3 constraint, but in consultation with Patrick decided to concentrate on GBAC constraints for the rest of the project.

16th October 2006 Implemented Um-3 toolkit constraint.

23rd October 2006 Designed GBAC Um-3 constraint. Started implementation.

30th October 2006 Presented design of constraint to Patrick and Chris. Outlined proof of correctness and discussed it. Continued implementation.

6th November 2006 Proved constraint correct. Needed difficult lemma that clipped inf can lead to clipped sup, but not the converse. Started designing Um-4 based on this.

13th November 2006 Started coding Um-4 constraint.

20th November 2006 Project on hiatus for assessed coursework.

27th November 2006 Project on hiatus for assessed coursework.

4th December 2006 Completed design, proof of correctness and implementation of UM-4 constraint.

11th December 2006 Investigated constraint using fact that UM matrices have fewer than $n-1$ distinct variables, didn't finish work because constraint seemed likely to offer weak propagation.

18th December 2006 Designed and coded WAIT-2 and WAIT-3 constraints as straw men.

25th December 2006 Project on hiatus for Christmas and general private study.

1st January 2007 Project on hiatus for New Year and general private study.

8th January 2007 Re-read AC-5 paper. Identified my constraints as being suitable for AC-3and AC-5 propagators. Carefully designed a matrix AC-3 propagator.

15th January 2007 Implemented MAT-UM-3 and MAT-UM-4 constraints.

22nd January 2007 Ran preliminary experiments on species tree data. Noticed phase transition behaviour varying in different constraints.

29th January 2007 Identified flaws in experiment to do with lack of knowledge on error bounds and inconsistent environment.

5th February 2007 Briefly read literature on random empirical studies in constraint programming, kappa, etc. Decided to concentrate on testing instances of roughly same difficulty instead of doing full phase transition investigation, due to time constraints. Worked out idea of using two constraints at once to exploit hypothesised difference in phase transition behaviour for different constraints. Designed random empirical experiment.

12th February 2007 Wrote shell scripts to execute and produce tables and graphs for random empirical study. Ran experiment, noticed strange behaviour and wrote report section summarising results.

19th February 2007 Discussed with Patrick explanation for why variable ordering heuristics make such a big difference. Planned experiments on real species data.

26th February 2007

5th March 2007 Ran experiments on real life species trees. Analysed results and wrote report section summarising results.

12th March 2007 Project on hiatus for assessed coursework.

19th March 2007 Report writing.

26th March 2007 Report writing.

2nd March 2007 Redrafting of report and preparation for presentation.

9th March 2007 Final check over report, presentation and submission.

Acknowledgements

"Friendship is one of the most tangible things in a world which offers fewer and fewer supports."— **Kenneth Branagh**

Thanks to Patrick Prosser for introducing me to constraint programming and giving me the problem to solve. I've always needed to be pointed in the right direction and I think that he's given me a long enough road to traverse that I should have something interesting to do for the next few years.

Thanks to Chris Unsworth for his helpful comments in the early stages of the project.

Thanks to my friends and family. Thanks to writers of good books and discoverers of good ideas for inspiring me.

Bibliography

[BZF04] C. Bessiere, B. Zanuttini, and C. Fernandez. Measuring search trees. In B. Hnich, editor, *Proceedings ECAI'04 Workshop on Modelling and Solving Problems with Constraints, Valencia, Spain*, 2004.

[CKT91] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proc. of the 12th IJCAI*, pages 331–337, Sydney, Australia, 1991.

[Daw06] Richard Dawkins. *The Blind Watchmaker*. Penguin, 2006.

[DDD05] Grégoire Dooms, Yves Deville, and Pierre E. Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In *CP*, pages 211–225, 2005.

[DW04] Richard Dawkins and Yan Wong. *The Ancestor's Tale*. Weidenfeld and Nicholson, 2004.

[Fel78] Joseph Felsenstein. The number of evolutionary trees. *Systematic Zoology*, 27:27–33, 1978.

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.

[GPSW03] Ian P. Gent, Patrick Prosser, Barbara M. Smith, and Wu Wei. *Supertree Construction with Constraint Programming*, pages 837–841. Principle and Practice of Constraint Programming. Springer, 2003.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[HDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992.

[HE80] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980.

[KP02] M. Kennedy and R.D.M. Page. Seabird supertrees: Combining partial estimates of procellariiform phylogeny. *The Auk*, 69:88–108, 2002.

[Mac75] Alan K. Mackworth. Consistency in networks of relations. Technical report, Vancouver, BC, Canada, Canada, 1975.

[Mic07] Sun Microsystems. Javadoc for `System.gc()`. `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html#gc()`, February 2007.

[NW96] Meei Pyng Ng and Nicholas C. Wormald. Reconstruction of rooted trees from subtrees. *Discrete Appl. Math.*, 69(1-2):19–31, 1996.

[Ols90] Gary Olsen. Interpretation of the "newick's 8:45" tree format standard. `http://evolution.genetics.washington.edu/phylip/newick_doc.html`, August 1990.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Pro06] P. Prosser. Supertree construction with constraint programming: recent progress and new challenges. In *WCB06 - Workshop on Constraint Based Methods for Bioinformatics*, pages 75–82. N/A, 2006.

[PSW00] P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. *Proceedings of CP 2000: the 6th International Conference on Principles and Practice of Constraint Programming, (lecture Notes in Computer Science)*, pages 353–368, 2000.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[Ros98] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 1998.

[SF94a] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, volume 874, pages 10–20, 1994.

[SF94b] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA*, volume 874, pages 10–20, 1994.

[Smi01] Barbara M. Smith. Lecture notes in constraint satisfaction and constraint programming. `http://www.dcs.gla.ac.uk/~pat/cp4/contrib/bms/AR33.pdf`, 2001.

[Var07a] Various. Megabyte. `http://en.wikipedia.org/wiki/Megabyte`, April 2007.

[Var07b] Various. Speciation. `http://en.wikipedia.org/wiki/Speciation`, February 2007.

[Wal93] R. J. Wallace. Why ac-3 is almost always better than ac-4 for establishing arc consistency in csps. In *Proc. of the 13th IJCAI*, pages 239–245, Chambery, France, 1993.

[Wei07a] Eric Weisstein. Metric. `http://mathworld.wolfram.com/Metric.html`, February 2007.

[Wei07b] Eric Weisstein. Ultrametric. `http://mathworld.wolfram.com/Ultrametric.html`, February 2007.

[YY01] Zhang Yuanlin and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *IJCAI*, pages 316–321, 2001.

Index